1993

Integration of object-oriented and relational database systems

Seyed Mohsen Sedighi
University of Wollongong

INTEGRATION OF OBJECT-ORIENTED AND RELATIONAL DATABASE SYSTEMS

A thesis submitted in partial fulfilment of the

requirements for the award of the degree

Master of Science (Honours)

(Computer Science)

from

THE UNIVERSITY OF WOLLONGONG

by

Seyed Mohsen Sedighi, BSc Comp. Sci.

DEPARTMENT OF COMPUTER SCIENCE 1993.

This is to certify that this thesis has not been submitted for a degree in any other University or Institution.

Abstract

In an organisation, there are many autonomous database systems with different data models, query languages, and schemas. The user in the organisational level needs to have a shared access to the databases by using a single data model, and a single query language.

The aim of this thesis is to propose and evaluate a method for integration of autonomous, heterogeneous database management systems (DBMSs) to facilitate shared access through a system, based on the method, called Database Integration Methodology (DIM).

The thesis first reviews the basic concepts as related to heterogeneous DBMSs, then DIM is proposed as a general method for integration of a number of heterogeneous DBMSs. A DIM system has one unified data model to give a homogeneous view of the databases to the user. It has also a database query language for retrieving and manipulating data from the databases. We present a method and explain through it, how a translated query language statement of a DIM system which is a set of query language statements of local database systems, is combined to make a shared access to the databases possible.

As one particular case of the general method, DIM is implemented for integration of object-oriented and relational DBMSs. We choose relational data model as the unified data model of the DIM. A solution for translation of object-oriented schemas into relational schemas is considered.

Acknowledgment

I would like to extend my sincere thanks to my supervisor Dr. J. R. Getta without whose invaluable assistance this thesis would not have been possible.

I am also indebted to Associate Professor N.A.B. Gray for his recommendations of source books in the area of object-oriented paradigms.

My thanks also go to the technical staff in the department of computer science, for the help they rendered me.

Last and not least, I am grateful to my wife and children for their patience throughout this work.

Contents

3

An Overview of Relational and Object-Oriented DBMSs 16

ORACLE, and Pro*C are registered trademarks of Oracle corporation.

ObjectStore is trademark of Object Design Inc.

Chapter

1

Introduction

This chapter reviews the basic concepts of heterogeneous database management systems (HDBMSs). It also discusses specification of the project, and clarifies the meaning of database integration.

1.1 Review of the Basic Concepts

New applications require new database systems that can support solutions for the applications. In an organisation, there are different database systems with different data models; each resolves various application requirements. Generally, the preexisting database systems in the organisation can not be replaced with new database systems because it is costly, and is simply not feasible [23]. They continue to work and perform the specific application requirements of every department. The need to access all of these databases as one database with a single data model and a single query language leads to the problem of management of heterogeneous database systems.

Heterogeneous files are the files that contain redundancies, and for similar data they have various types of heterogeneity such as value types, differences in naming, and file structures. In the traditional data processing systems, there are many heterogeneous files created by separate applications in an environment. The heterogeneous files have many problems. For example, it is difficult to provide consistency, overall privacy and efficiency among the files[23].

To overcome these difficulties, the heterogeneous files were replaced by a centrally defined database which was free of heterogeneity, to some extent free of redundancies, and inconsistencies. The database was managed under the central control of a database management system (DBMS).

A DBMS is a software system for managing a set of data that is called database. The data in a database is organised according to a data model. The user accesses the data through an interface called the query language which is provided by the DBMS. A schema describes the actual data structure and organisation within the system.

A distributed database system (DDBS) has a logical database that is physically distributed and a distributed database management system (DDBMS) provides consistent queries and updates among the distributed databases (DDB). Furthermore, in a DDBS all its physical components run the same DDBMS that implies homogeneity. The DDBS has just one data model, one query language, and its schema is explicit.

A heterogeneous database management system (HDBMS) can be conceived as a distributed database system that has heterogeneous database (HDB) components, and their DBMSs are autonomous, i.e. they are independent of each other (Figure 1). The

heterogeneity can be different computer architectures, operating systems, data models, DBMSs, database schemas, and database query languages.

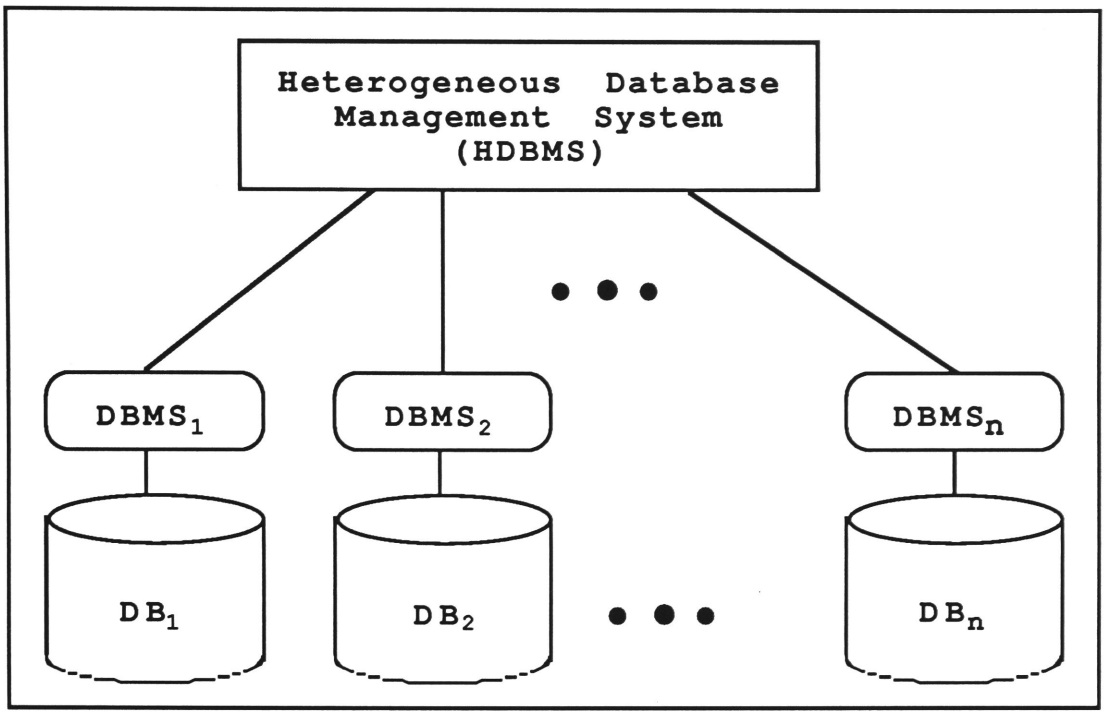

Figure 1. A Heterogeneous Database Management System (HDBMS)

Today, there are many database systems with different data models. With the advancements in distributed computing and networking, a shared access to these databases is required by the users. It seems that the same scenario for accessing heterogeneous files is repeated for accessing heterogeneous databases. This time, of course, with more problems and difficulties. However, full cooperation of the database systems is possible, while their interoperation is not feasible yet.

Other terms related to HDB include federated databases, multidatabases, interoperable databases, and interdependent databases. Many authors have written different definitions for these terms. In practice they are used as synonymous [23]. For this project we choose a new term called, Database Integration Methodology (DIM) to avoid a specific meaning of the above terms.

1.2 Specifications of the Project, and Its Aims

Our task is to use a methodology for integration of a number of autonomous, heterogeneous databases of which relational, and object-oriented databases are the particular case. The databases operate independently of each other. They are different in respect to their data models, query languages, and schemas.

The project investigates how it is possible to integrate heterogeneous databases without affecting their autonomy. This makes shared access to the databases possible by using a single data model and a single query language. The integration means that the databases logically become a homogeneous database for the users. Thus, it hides the heterogeneity of data models, query languages, and schemas of the databases so that the users view them as a single database. The aim is syntactical integration of the databases. Semantic heterogeneity of them is not investigated in this integration. For example, it does not consider if and how two or more objects in different databases are related.

As a general method, integration of several heterogeneous databases is achieved through a method, called Database Integration Methodology (DIM). DIM is a general method that one particular case of it, is implemented for integration of object-oriented and relational DBMSs. A system based on DIM (from now on we call it DIM system) has a unified data model, and a query language. In the particular case, the unified data model is selected to be relational modelling of data, and the SQL be the query language. We consider a solution for translation of object-oriented schemas into relational schemas. A DIM system presents the user several schemas that some of them are as the result of the translation. Not only query statements can be made against the databases, but also update commands as well. Preserving the autonomy of the databases allows execution of local

applications to be continued without any changes (Figure 2). A DIM system has no global transaction management system, and relies on transaction management of local database systems.

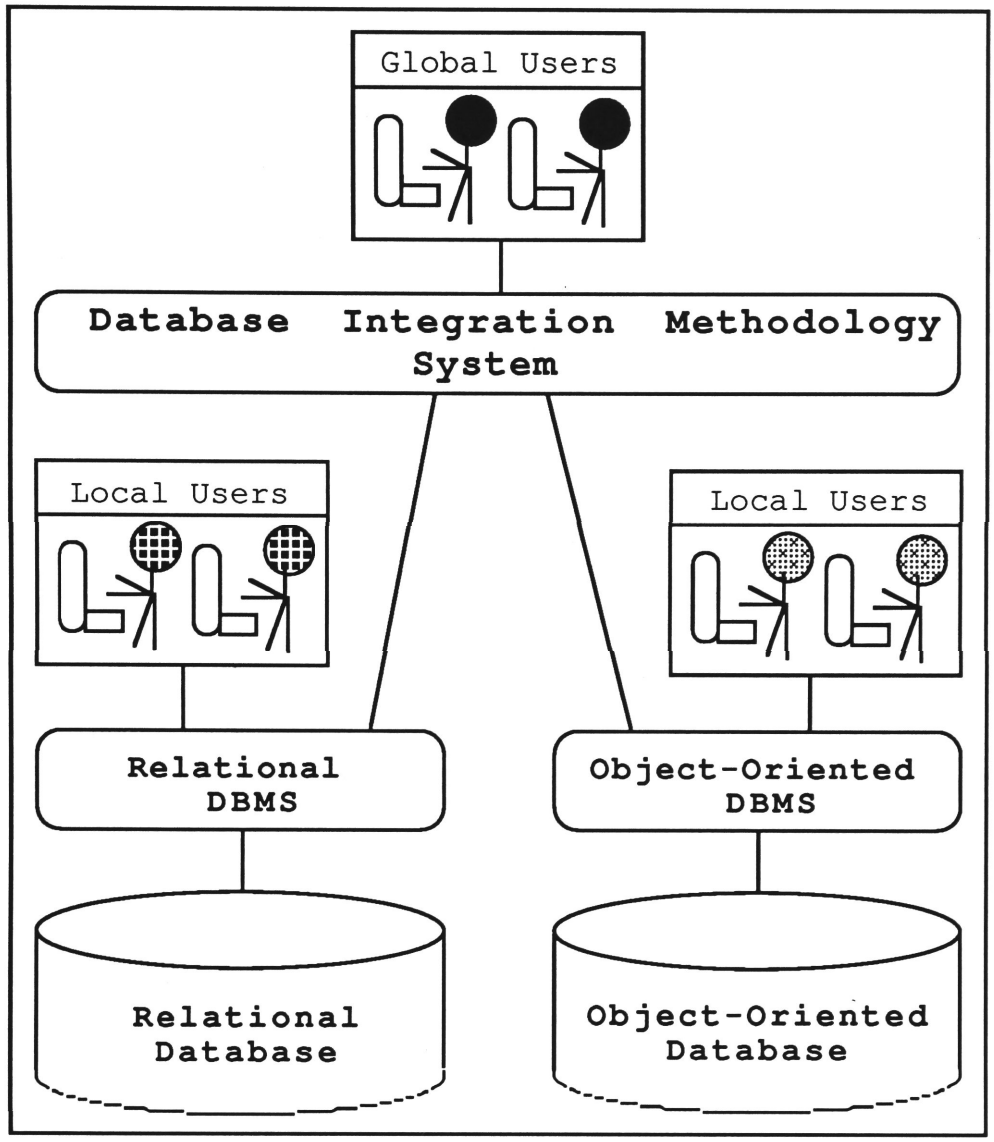

Figure 2. Local users can access one database, relational, or object-oriented. Global users can concurrently access both databases through DIM system.

1.3 Summary

This chapter discussed the basic concepts of HDBMSs, and the aim of the project. An important feature of a HDBMS is to hide the heterogeneity of participating databases from the user, while preserving the autonomy of them.

The organisation of subsequent chapters are as follows. Chapter 2 deals with heterogeneous database systems in detail. In chapter 3, we review the basic concepts as related to object-oriented, and relational DBMSs. Chapter 4 examines comparison of object-oriented and relational database systems. In chapter 5, DIM is introduced, and integration of object-oriented and relational DBMSs is addressed. Chapter 6 outlines implementation of DIM in the case of object-oriented and relational DBMSs. Finally, the last chapter is designated to the conclusions.

Chapter

2

An Overview of HDBMSs

A HDBMS is a system which provides a shared access to database systems with different data models. The databases are managed under their DBMSs.

2.1 Motivation

Typically, in an organisation, each department has its autonomous database which provides its own information needs by means of specific database applications. For example, the personnel department may use a record-oriented database such as relational database system, to keep track of the employee records. The engineering department may use an object-oriented database, for design specifications in terms of product assemblies. Yet another department may use other database model for its application requirements and so forth. These database systems may be on main frames, mini and micro computers. In the department level, the databases, data languages, and database systems, with respect to

the specific database applications of that department, are homogenous. In the organisation level they are heterogeneous.

Shared access to the dispersed information in different departments is necessary due to corporate planning, decision-making, marketing strategies, and other multi-departmental activities. Users in one department should be able to have a shared access to the databases of other departments without learning data models and query languages of the multiple databases.

2.2 Characteristics

Heterogeneous database systems typically integrate information from preexisting local databases in an enterprise and present global users with transparent routines to use the total information in the system. Important dimensions associated with these systems are data replication, heterogeneity, and the autonomy that individual databases retain [32].

2.3 Data Replication

Data may be replicated among local databases. For example, multiple copies of some or all of the data may be maintained in the local databases with different structures. Data replication has many benefits such as increased availability and reliability as well as improved access times. In contrast to the benefits, data replication causes data redundancies and inconstancies. Much of the data replication may be due to the existence of local databases before any heterogeneous database system is built.

2.4 Heterogeneity

Heterogeneity may be due to technological differences such as differences in hardware, system software, and communication systems. The types of heterogeneity in the database systems are the differences in DBMSs and the semantics of data.

2.4.1 Heterogeneity due to Different Database Systems

In an enterprise, there are different DBMSs which have been purchased over a period of time because of new application requirements. Each DBMS has its own data model which is different from the others. Heterogeneities due to differences in DBMSs result from differences in data models.

Different data models provide different structural primitives. For example, the information modelled using a table in the relational model may be modelled as an object in the object-oriented model.

Constraints may be supported by two data models differently. For example, in the relational model, primary keys are used for identifying a tuple in a relation, where in the object-oriented model, object identity provides a unique identifier for recognising an object.

DBMSs based on different data models usually use different languages to manipulate data which lead to heterogeneity. For example, relational models generally

support SQL to access the databases but most object-oriented models use an extension of an object-oriented programming language to access the object-oriented databases.

Another sort of heterogeneity is the implementation aspects of DBMSs. Different transaction primitives such as concurrency control, and recovery are of this type.

2.4.2 Semantic Heterogeneity

Semantic heterogeneity appears when two objects that represent the same real world entity have different information [31]. Semantic heterogeneity may also refer to a disagreement about the meaning, interpretation, or intended use of the same or related objects. Differences in data definition of the same facts are called conflicts [8]. Conflicts are classified as follows:

* Name conflicts: Local databases may have different conventions for naming objects that results in conflicting names of those objects. Name conflicts typically involve homonyms and synonyms. Homonyms means the same name is used for different facts, and synonyms is to use different names for the same fact.

* Format conflicts: Differences in data type, and precision leads to format conflicts. For example, an employee number may be defined as an integer in one database, and an alphanumeric string in another.

* Scale conflicts: The use of different units of measure called scale conflicts. For example, in one database degrees of Fahrenheit may be used for measurement, and Celsius degrees in another.

* Structure conflicts: Two databases may use different structures for the same facts. For example, in one database an attribute may be used to define an address, and in the other database many attributes construct an address.

* Missing or different information conflicts: The same object in different databases may have different recorded values due to incomplete update or the lack of availability of the related information.

2.5 Autonomy

Database systems are often under separate and independent control. Several types of autonomy including design, communication, execution, and association autonomy are discussed in [32]. Heterogeneity in a HDBS is originated by design autonomy among local database systems.

Design autonomy refers to the ability of a local database system to choose its own design in regard to any subject, including data model, query language, naming of the data elements, semantic interpretation of the data, constraints, sharing with other systems, and the implementation. Communication autonomy refers to the capability of a database system to decide whether to communicate with other database system.

Execution autonomy is the ability of a local database system participating in a HDBS to execute local operations without obstruction from operations submitted by the other database systems and to decide the order in which to execute the submitted

operations. Operationally, a local database system, performs its execution autonomy by treating the submitted operations in the same way as local operations.

Association autonomy refers to the ability of a local database system to decide whether and how much to share its functionality and resources with others in the HDBS. Association autonomy requires that each local database system be free to associate or separate itself from the HDBS. This would necessitate that the HDBS be designed so that its existence and operation not to be dependent on any single database system.

2.5.1 Advantages and Disadvantages

Preserving autonomy of local database systems in the HDBS, has many advantages and disadvantages [5]. Advantages of the autonomy are:

* A local DBMS can be incorporated into the heterogeneous system without affecting its local functions.

* For the users of a local DBMS nothing will change, when the local DBMS join a heterogeneous system.

* The autonomy can act as a security measure because the local DBMS has full control over its functions. It is the local DBMS that allows who can access local resources.

In spite of the advantageous aspects of the autonomy, it has many disadvantages which are as follows:

* The autonomy puts a large responsibility on the global database administrator (GDBA).

* Global requirements and global optimisations are likely to conflict with local ones.

* Some global standards may be enforced to local DBMSs by the local DBAs.

2.6 Loosely Coupled and Tightly Coupled

Classification of HDBSs to loosely and tightly coupled systems depends on how much local DBMSs are autonomous in relation to the HDBS managing them. In the most loosely coupled system, access to local databases is through the local DBMSs user's interface. Consequently, the local databases completely uphold their autonomy.

In the most tightly coupled system, access to internal functions of the local DBMSs are possible by the HDBS. This allows synchronisation among the underlying DBMSs and efficient global processing, possibly at the expense of local efficiency. As a result, local DBMSs have not full control over local resources.

2.7 Design of HDBMSs

In the database literature, there are many approaches to design heterogeneous database systems. Two major approaches are global or unified schemas and multidatabase languages [5][28].

In the global schema approach a data model is favoured as a common data model (CDM) for unification of the underlying data models. Then, a unified schema of the constituent local databases is defined in the CDM by the global database administrator (GDBA). Therefore, global users view a single integrated database which is free of heterogeneity. The global schema is just another layer above the local schemas (Figure 3) and it is usually replicated at each node for efficiency. For specific users and applications, views may be defined on the top of the global schema. In this approach the GDBA has many responsibilities such as resolving semantic and syntactic differences among local schemas and creating an integrated summary of their information.

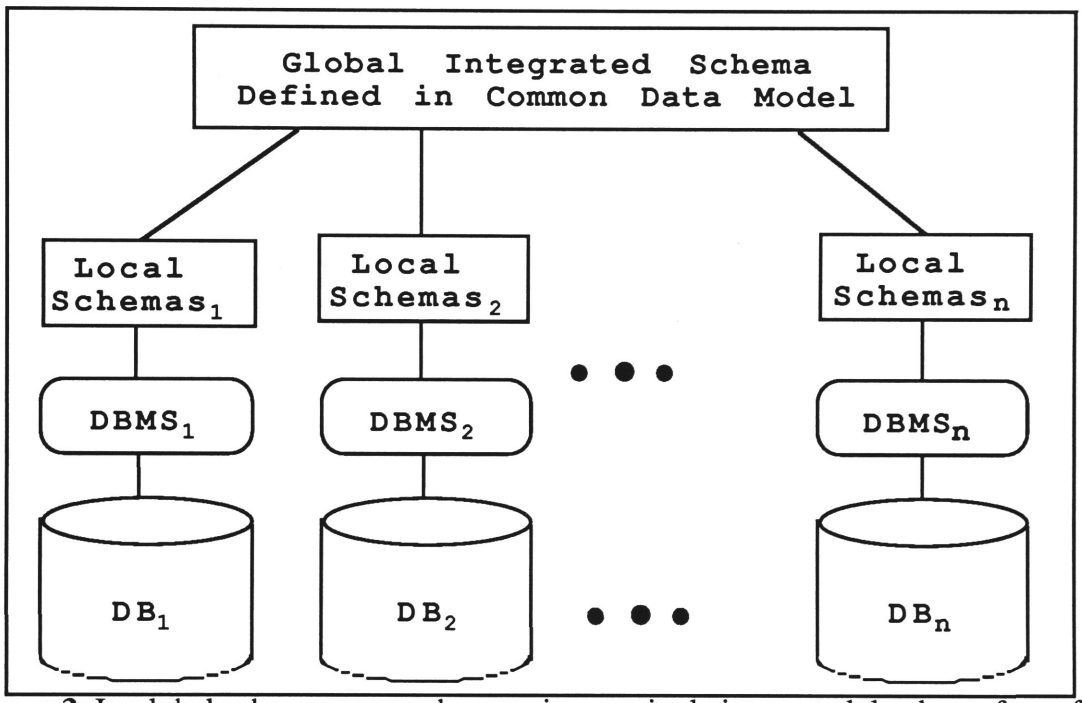

Figure 3. In global schema approach user views a single integrated database, free of all heterogeneity in local DBMSs.

In the multidatabase language approach, complete integration is not performed to preserve the autonomy of the local databases. Therefore, this approach has no single integrated schema and puts most of the integration responsibility on users. Instead, it eases the burden of users' tasks by giving them many support functions, and by providing more control over the information. The functions are added to database languages for data

manipulation. In this approach, users must have some global knowledge of representation differences and data resources, but only about the information actually used.

2.8 Summary

Shared access to heterogeneous databases in an organisation is inevitable. A HDBMS typically integrates information from preexisting databases, and presents global users with transparent methods in order to use the total information in the system. Important dimensions associated with these systems are data replication, heterogeneity, and the autonomy that individual databases retain.

The types of heterogeneity in the database systems are the differences in DBMSs, and the semantics of data. Heterogeneity in a HDBS is originated by design autonomy among local database systems. Database systems are often autonomous. In spite of disadvantageous aspects of the autonomy, preserving the autonomy of database systems in the HDBMS has many benefits. Classification of HDBSs to loosely and tightly coupled systems depends on how much local DBMSs are autonomous in relation to the HDBS managing them. Two major approaches to design heterogeneous database systems are global or unified schemas and multidatabase languages.

Chapter

3

An Overview of Relational and Object-Oriented DBMSs

3.1 Relational DBMSs

Primary appeal of the relational model is its simplicity. The relational model has only one data structure that is called table or relation. The operators of the model operate on such tables to produce new tables or relations. There are only two inherent integrity constraints in the model which are known as entity and referential integrity [3].

The relational model has theoretical foundation based on the mathematical notion of relation [2]. Tables are a natural representation. Data is represented by means of values, and relationship between data is not represented explicitly. Further, relationship between data is established by means of values. Data independence is an important property of the relational model, allowing changes in data organisation without modifying application programs.

3.1.1 Domains, Attributes, Tuples, Relations, and Keys

A domain D is a set of atomic values, which must be of the same data type. The set of character strings, the set of integers, real numbers, and the set {0,1} are examples of a domain. A relation schema R, denoted by $R(A_1,A_2,..., A_n)$, is a set of attributes $R = \{A_1,A_2,..., A_n\}$. Each attribute A_i is the name of a role played by some domain D in the relation schema R [12]. D_i is called the domain of A_i and can also be denoted by $dom(A_i)$.

A relation r of the relation schema $R(A1, A2,..., A_n)$ denoted by r(R) is a subset of the Cartesian product of the domains that define R. The Cartesian product of domains D_1, D_2, ..., D_n, denoted by $D_1xD_2x...D_n$, is the set of all n-tuples $(v1, v2,..., v_n)$ such that v_1 is in D_1, v_2 is in D_2, and so on[36]. For example, if k = 2, D_1 = {0,1}, and D_2 = {a, b}, then the Cartesian product of D_1, D_2 is :

D_1xD_2 = {(0,a),(0,b),(1,a),(1,b)}

Any subset of the Cartesian product D_1xD_2 is a relation. The members of a relation are called tuples. A tuple $(v1, v2,..., v_n)$ has n components.

In a relational database, each relation is represented by a table, where each row is a tuple, and each column corresponds to one component. Attributes are the names of columns. Date [9] has described formal and informal relational terms which are given in Table 1.

A relation is defined as a set of tuples. Since all elements of a set are distinct, all tuples in a relation must also be distinct. A relation has sets of one or more attributes that serve as keys. A set S of attributes of a relation r of a relation schema R is a key if[36]:

1. No instance of r(R) can have two tuples that can agree in all the attributes of S, yet are not the same tuple,
2. No proper subset of S has property (1).

When a relation has two or more keys, one of them is selected as the only key. The term primary key is used to refer to the key selected from among several choices, all of which are called candidate keys.

Primary keys are of high importance to the relational data model. In combination with the table name, a primary key value provides a sole addressing mechanism which is the only quaranteed way of locating a given row from a relational database. This is the reason that the relational data model is often referred to as a value-based data model.

Formal Relational Terms	Informal Equivalents
Relation	Table
Tuple	Row
Cardinality	Number of Rows
Attribute	Column
Degree	Number of Columns
Primary Key	Unique Identifier
Domain	Pool of Legal Values

Table 1. Formal and informal relational data structure terms

Foreign keys are the means of interconnecting the information stored in a series of disparate tables. Formally, a set of attributes FK in relation schema R1 is a foreign key of R1 if it satisfies the following two rules[12]:

1. The attributes in FK have the same domain as the primary key attributes PK of another relation schema R2,
2. A value of FK in a tuple t1 of R1 either occurs as a value of PK for some tuple t2 in R2 or is null.

3.1.2 Data Manipulation

There are two rather different kinds of notations used for expressing operations on relations [36]:

1. Algebraic notation called relational algebra.

2. Logical notation called relational calculus.

In relational algebra, queries are expressed by applying specialised operators such as JOIN, PROJECT, etc. In relational calculus, queries are expressed by writing logical formulas that the desired relation in the answer must satisfy them. The algebra is procedural, because it requires the specification of the way the result is computed, whereas the calculus is declarative or non procedural, because it requires the description of the properties of the results.

Codd in his early paper (1970) proposed a collection of operators for manipulating relations and called these operators the relational algebra. The relational algebra is a set of eight operators. Each operator takes one or more relations as input and produces one relation as output (Figure 4).

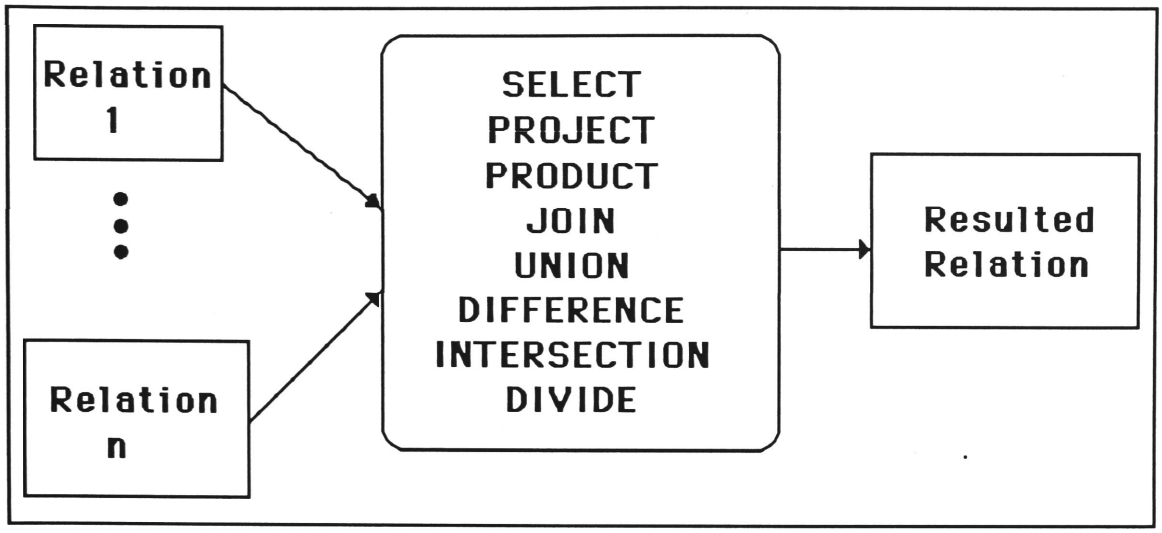

Figure 4. Algebraic operators act on relation(s) to produce a new relation.

The three main operators of the algebra are SELECT, PROJECT, and JOIN which are the basic data manipulation of relational systems. The operators: PRODUCT, UNION, DIFFERENCE, INTERSECTION, AND DIVIDE are modelled on the traditional operators of set theory.

3.1.3 Integrity Constraints

The relational model has two inherent integrity constraints. They are entity and referential integrity. Other types of constraints are Key, domain, and semantic integrity constraints of the relational model [12].

The entity integrity constraint concerns primary keys. It states that every relation must have a primary key and no component of the primary key is allowed to accept nulls.

The referential integrity constraint concerns foreign keys. It states that the database must not contain any unmatched foreign key values. Occasionally a foreign key value can

be null. In this case, there is no relationship between the objects represented in the database or this relationship is unknown.

The key constraints specify the candidate key values of each relation must be unique for every tuple in that relation. The domain integrity constraint enforces the set of legal values for each domain. The semantic integrity constraints, specify the semantic of a relational database, that may need to be specified and enforced on the database. An example of such a constraint is " the salary of an employee should not exceed the salary of the employee's supervisor" [12].

3.1.4 Normalisation

There are a number of problems that normalisation is intended to avoid [35][36]. The problems are redundancy, and different kinds of anomalies in a database that are explained in the follows:

* Redundancy: The same information be present in the database in several physically different locations.

* Potential inconsistency (update anomalies): If, for example, a person's age is stored several times, after a series of invalid updates, the database could represent the person as having several different ages.

* Insertion anomalies: There is no way of storing entities without storing some associated entity.

* Deletion anomalies: It is the inverse of preceding anomalies, i.e. entities are deleted if some associated entity is deleted.

Normalisation is a step-by-step process for breaking up larger tables into smaller tables to satisfy a certain set of constrains that is called normal forms. A series of normal forms has been defined in the database literature to prevent the above mentioned problems. The following example is taken from [36] to show the problems.

Suppose a relational schema is:

SUPPLIER_INFO (SUP_NAME, SUP_ADDRESS, ITEM, PRICE)

that included all the information about suppliers. The problems with this schema are:

* The Address of the supplier is repeated once for each item (redundancy).
* The address for a supplier in one tuple could be updated, while leaving it fixed in another (potential inconsistency).
* An address for a supplier cannot be recorded if that supplier does not currently supply at least one item (insertion anomalies).
* If all of the items supplied by one supplier is deleted, track of the supplier's address will be lost unintentionally (deletion anomalies).

All the above problems go away if SUPPLIER_INFO is replaced by the two relational schemas:

SUPPLIERS (SUP_NAME, SUP_ADDRESS)

SUPPLIES (SUP_NAME, ITEM, PRICE).

3.1.5 The Database Sub-language SQL

Operations on the database require a specialised language that is called query language. The structured query language (SQL) was originally designed as a query language based on the relational algebra.

A significant feature of SQL as implemented in a number of commercial database systems is that the same language is available at two different interfaces. One is an interactive interface and the other is an application programming interface. Therefore, SQL is both an interactive query language and a database programming language. Any SQL statement that can be entered at a terminal can be embedded in a program.

SQL has three major parts:

1. A data definition language (DDL).

2. A data manipulation language (DML).

3. A data control language (DCL).

The major data definition functions are:

CREATE TABLE
CREATE VIEW
CREATE INDEX
DROP TABLE

DROP VIEW,
DROP INDEX.

The major data manipulation functions are:

SELECT
UPDATE
INSERT,
DELETE.

The main data control functions are:

GRANT,
REVOKE.

3.1.6 Features of SQL

SQL has many features including [7][14]:

* SQL is a standard query language of database systems.

* SQL provides users with the formulation of ad-hoc queries in a natural way.

* SQL is one of the simplest languages for aggregate manipulation.

* In SQL, the user specifies what is to be accessed from the database rather than how to access it.

3.1.7 Host Languages

Manipulation of the database can be done by an application program that is written in advance to perform a certain task. Sometimes, it is necessary for an application program to perform a variety of computational task in addition to manipulate the database. Programs that manipulate the database are commonly written in a host language, which is a conventional programming language such as C, PASCAL, COBOL, or PL/I. The host language is used for everything but the actual querying and modification of the database [36].

Depending on the characteristic of the DBMS, the command of the data manipulation language are invoked by the host language program in two ways. One way is that the command are invoked by host language calls on procedures provided by the DBMS. Another way is that the commands are statements in an extension of the host language. In this case a preprocessor converts the statements into calls on procedures provided by the DBMS (Figure 5).

When a database query language such as SQL is used in an application program written in a host language, it causes a sort of type incompatibility called the impedance mismatch. This is because programming languages are procedural, and database languages are higher-level and more declarative. Furthermore, the data types in the different languages i.e. SQL and the host language, are not the same and have to be mapped onto one another.

When relational languages are embedded in host languages, some mechanism is provided to couple the two languages. In the case of SQL, cursors are supplied. A cursor

can be conceived as a tuple variable that ranges over all the tuples of the relation that is the answer to a query.

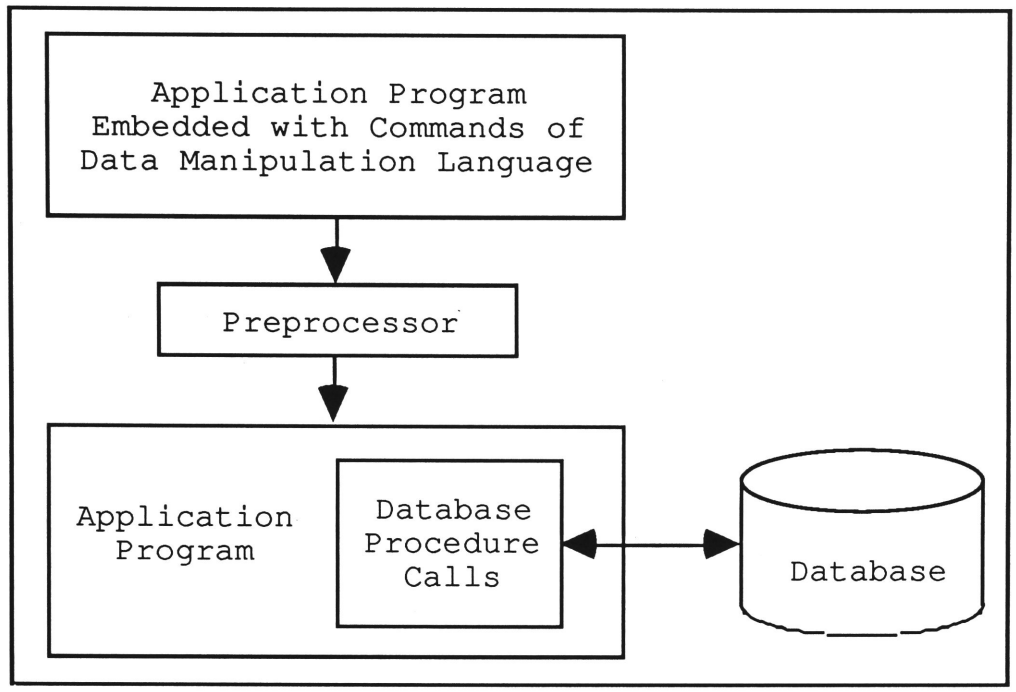

Figure 5. Conversion of database commands in an application program into database procedure calls.

3.1.8 Limitations of RDBMSs

Relational models often reduce application development time, and offer good storage management capability. However, they are subject to many limitations [19] which are summarised in the following [16].

* Because of restriction in modelling power of relational systems, some objects (e.g. complex objects) cannot be adequately represented.

* There is only a fixed set of operations for atomic data values and tuples. Moreover, it is not possible to add new operations and make those operations appear the same as the built-in operations.

* The systems are not suitable for complex design applications such as CAD/CAM, and CASE.

3.2 Object-Oriented DBMSs

Object-oriented database management systems (OODBMS) aimed at supporting engineering, scientific, and office information applications. They provide more sophisticated generations of traditional business applications.

OODBMSs have been built with a variety of data models and implementation architectures. There is no single commonly accepted data model for OODBMSs yet. Object-oriented databases are rooted in the same concept as object-oriented languages. Essentially, most OODBMSs are based on extension of existing object-oriented programming languages which provide persistence, concurrency control, and other database capabilities. Generally, they are called database programming languages. In an OODBMS, there is a seamless integration between the programming language and the database.

3.2.1 Objects

Objects are the elements that an object-oriented database system stores, modifies, and retrieves. Every object do have an identifier, which is a unique system generated

identifier, and is unique over all time. The identifier is as a pointer to a persistent object. After creation of an object, it can persists until there will not be any need for its existence. In fact, there are two kinds of objects [1]:

1. Volatile objects which are allocated in volatile memory and are the same as those created in ordinary programs. Volatile objects' lifetime is bounded by the life of the program creating them.

2. Persistent objects are located in persistent store. They continue to exist after the program that created them has terminated.

3.2.2 Complex Objects*

Complex objects are objects that have hierarchical structure and the value of their attributes can be objects themselves. Simplest objects are integers, characters, and other simple types. Complex objects are built by using constructors to the simple objects. In this way, sets, lists, bags, and arrays can be built. An important requirement to complex objects is that it should be possible to create, manipulate and access them in their entirety [11].

3.2.3 Classes or Object Types

A class is an abstract data type that consist of a data structure definition and a collection of operations, called methods that apply to objects in the class. Classes provide

* The term "complex values" can also be used instead of "complex objects".

the facilities to define well-encapsulated object types with their operations. Encapsulation refers to the ability to make parts of the implementation details inaccessible to the user. Classes are templates from which objects may be created. Classes support classification by allowing groups of objects to be created which share identical behaviour.

In object-oriented systems, the term class is the single most confusing term. The term has been used with more than one meaning [7] such as:

* A class defines an object type or the structure or behaviour of objects of a particular type.

* A class defines the set of objects having a particular type that is sometimes referred to as a classification of objects.

* A class defines a representatives, i.e. types are represented by objects themselves, so the class is an object, or meta object.

3.2.4 Class Extensions

The extension of a class is the actual instances of the class that have been created but not destroyed, i.e. they outlive. This extension then becomes a persistent collection of objects [20].

In object-oriented programming languages such as Smalltalk or C++, the user defines a class expressively as a template to generate objects. These languages provide the

primitives to create objects and implicitly or explicitly destroy the objects but they have no support for class extensions.

The most important use of class extensions is in database systems which process large numbers of objects of the same type. Several object-oriented database systems incorporate the notion of a class extension to make the instances of a class persistent. These systems provide different selection or iteration language primitives to associatively provide or update instances of a class.

3.2.5 Object Identifiers

In a complete object-oriented system, each object has an identifier that permanently is associated with it. The identifier is not related to the structure or value (state) of the object, and is maintained across multiple program or transaction instantiations.

There are many ways to distinguish objects from one another. One way is addressability, i.e. to use variable names. Addressability is external to an object and it provides a way to access an object within a particular environment. Another way to distinguish objects, is object identifier. Object identifier is internal to an object, and it provides a way to represent the individuality of an object independently of how it is accessed.

The use of variable names to distinguish objects is environment dependent and has practical limitations. For example, at an abstract level, a single object may be accessed in different ways and bounded to different variables without a mechanism to find out if they refer to the same object.

With object identifier it is simply possible to assign self-contained objects to attributes or instance variables and it is possible to make the same object a component of multiple objects [27].

3.2.6 Instance Variables

The internal representation of a class is captured by the instance variables, i.e. all the variables that hold the state of objects that are instances of the given class.

3.2.6.1 Complex Attributes

Object attributes or instance variables may have simple values such as integers, character strings, or other simple values. They may also have complex values, such as sets, arrays, lists, bags, or references to other objects. Moreover, users may also define their own value types to augment the built-in types, for instance, integers, or strings, by specifying the constructors on the type.

Reference attributes are used to represent relationships between objects. The values of them are object identifies for references to another objects. They are the same as pointers which can not be corrupted. They may be conceived as foreign keys in a relational system with following differences:

* Their values are not visible to the user.

* When the referenced object is deleted, reference values are automatically invalidated.

* The values of referenced object is not related to the reference attributes.

3.2.7 Relationships Between Two Objects

Relationship between two objects can be stored in different ways [4]. This is established via reference attributes of the objects. In other words, the values of reference attributes of one object are object identifiers of other objects. Different ways of relationship between two can be:

1. One object contains a reference attribute to another (Figure 6).

2. Both objects contain reference attributes to each other, called inverse attributes (Figure 7).

3. The relationship may be through a third object (Figure 8).

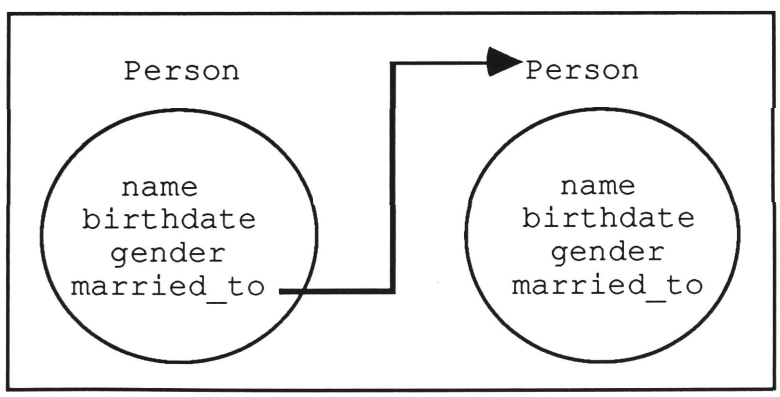

Figure 6. Representation of relationship between two objects when one of the objects has a reference to the other one.

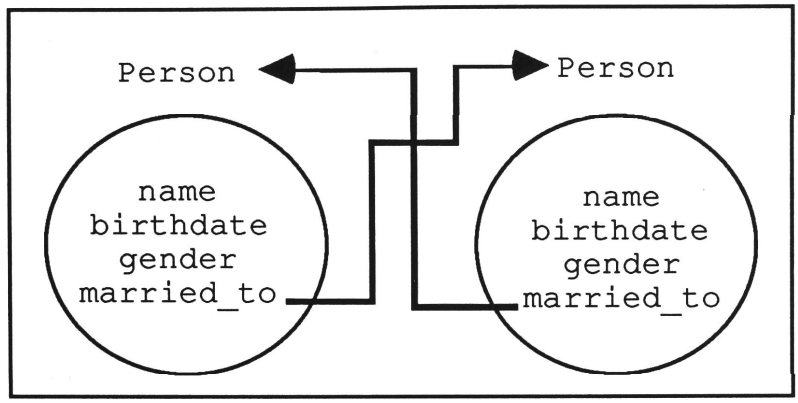

Figure 7. Representation of relationship between two objects when two objects contain a reference to each other.

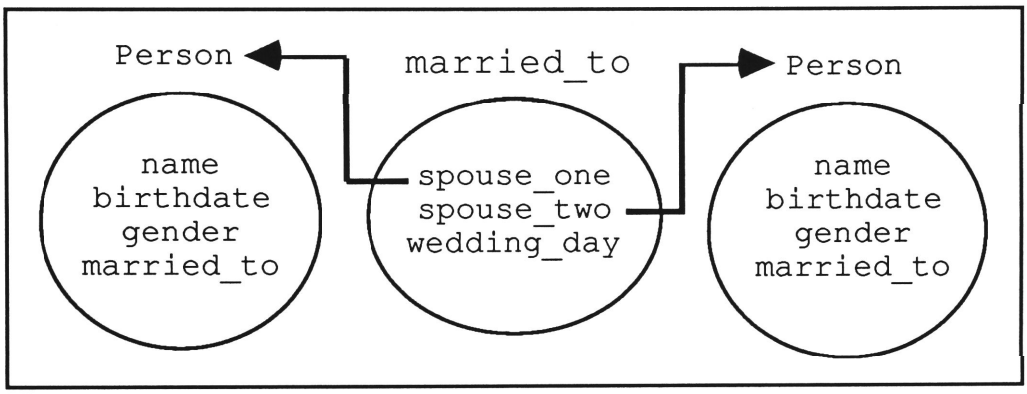

Figure 8. Representation of relationship between two objects through third object.

Relationships can be one-to-one such as each person having one spouse, one -to-many such as a person having many children, many-to-one such as many employees work in the same department, and many-to-many such as employees can work in different departments.

3.2.8 Inheritance

One of the main semantic abstraction is generalisation/ specialisation. The full benefits of object-oriented computing are realised when classes are extended by means of

the concept of inheritance. A new class can be described in terms of an existing class but perhaps with some changes. The new class, called subclass, inherits methods and data of the existing class, called superclass. The changes to the subclass, can be to add new operations and data, or to redefine existing operations to the new class. So, inheritance provides a very powerful mechanism that allows the sharing of data and operations.

When a new class is created from an existing class, the user creates a specialisation which is normally a step nearer the requirements of the application domain. Object-oriented systems supports different forms of specialisation.

3.2.9 Polymorphism and Dynamic Binding

In general, polymorphism means the ability to take more than one form. In an object-oriented language, a polymorphic references is one that, can over time, refer to instances of more than one class [22]. Because of the ability to refer to more than one class of object, a polymorphic reference has both a dynamic and static type associated with it.

Dynamic binding means the code associated with a given procedure call is not known until the instant of the call at run time. With static binding, a procedure call is known at compile time. The relative merits of static binding versus dynamic binding, reduce to the relative importance of efficiency versus flexibility[6]. Static binding is more efficient; dynamic binding is more flexible. In object-oriented systems, dynamic binding is associated with polymorphism and inheritance in that a procedure call associated with a polymorphic reference may depend on the dynamic type of that reference.

3.2.10 Schema Evolution

Database applications require considerable flexibility in dynamically defining and changing the database schema. Schema evolution may be modification of a class definitions or modification of the inheritance structure of classes by the user. OODBMSs may assist the user in handling the modifications. For example, when a new attribute is added to a class, the system may implicitly fix all the existing instances of the class to have the new attribute, or the user may be responsible for fixing those instances.

3.2.11 Versioning

Essentially, in various data-intensive application domains such as CAD/CAM systems, and office information, the ability to maintain more than one version of individual objects or an entire database is very important. In these applications, the same object undergoes multiple changes, and it is desirable to keep access particular previous states of the object.

3.2.12 Advantages of OODBMSs

OODBMSs improve the modelling power and integrity of systems. Some advantages of these systems are as follows [16]:

* All applications/tools have a common view of the data.

* Real world objects can be modelled by the object-oriented paradigm, which is a natural representation of objects.

* Inheritance and dynamic binding promote code reuse, extension, and modification.

* Complex objects allow the representation of structured or hierarchical data.

* When new components and devices are added to the database, old utilities continue working, meaning that code is generic and not dependent on any particular representation.

* Object schema definition and their operations to some extent constitute an up-to-date documentation for the system.

* Meta information is available to programs, because not only data but the schema is stored in the database.

* More meanings of data can be explicitly expressed through abstract data types, encapsulation and polymorphic routines.

* For the purpose of application programming, new data types are indistinguishable from system supplied types.

* Integrity of the database can be enforced by system constraint checking.

3.2.13 Limitations of OODBMSs

In spite of the powerful modelling of data, and many other features, OODBMSs do suffer from the following shortcomings [13]:

* There is no standard data model for OODBMSs. The lack of standards has contributed to the lack of useful tools for the user and developer of the system.

* OODBSs seem to have the long learning curve. They require users to have programming skills, and not just application knowledge. Object-oriented programming skills are harder to acquire than traditional imperative programming skills.

* Early commercial object-oriented systems are not implemented all features of a real DBMS.

3.3 Summary

A discussion of the various concepts as related to the relational and object-oriented DBMSs was covered. Besides the basic concepts of relational systems, the SQL language, and host languages were investigated. Host languages are important to the construction of a system for integration of heterogeneous databases that will be discussed later.

OODBMSs provide more sophisticated generations of traditional business applications. The most important features of these systems include object identity, encapsulated operations, type hierarchies with inheritance, complex objects and operations, polymorphism, dynamic binding, schema evolution, and versioning.

Chapter

4

Differences Between Relational and Object-Oriented DBMSs

One of the fundamental features of the relational model is the formal specification of the entire data model. There is no such unique definition for the object-oriented model. Many of the object-oriented data model definitions are no more than informal descriptions of system features. Moreover, there is no single group or source who provides a formal, uniform, and unique definition and specification of object-oriented data model.

This chapter examines differences between the relational and object-oriented DBMSs from the various terms and concepts as related to both systems.

4.1 Record-Based Versus Object-Based

Generally, based on the internal structure of data models, the models fall into two categories [26]:

* Record-based

* Object-based

They are both used to describe data at the conceptual and external levels.

Relational models are record-based. In these models, the overall structure and a higher level description of the implementation are specified.

Object-oriented data models are object-based. They support object identity, and object orientation concepts. They lack the means of logical structure specification, but provide more semantic substance by allowing the user to explicitly specify constraints on the data.

4.2 Value-Based Versus Identity-Based

In relational data model, primary keys are user-supplied scalar values which are used to construct an identifier key representing the identity of an object. The identifier has normally a meaning in the real world. For example, an identifier can be an employee number that uniquely identifies a tuple in a relation.

In an object-oriented data model, the system attaches a unique identifier to each object. The identifier is generated by the system and is independent of the attribute values of the object. The identifier has no meaning in the real world.

4.3 Relationships

The models generally use references as the basis to represent relationship. In the relational model, references are represented by using primary keys. Matching between foreign key values of a relation with primary key values of another relation, expresses relationships between tuples. It provides no implicit support for referential integrity and is a potential source of update anomalies.

Object-oriented databases, use the strong notion of object identity to represent relationships between objects. Relationships are represented by storing the object identifier of the related object. By storing object identifiers of two related objects, referential integrity is guaranteed (inverse attributes). Since object identifier is independent from the attribute values of the object, updates to the attributes have no effect on the object identifier. When an object is deleted from the database, it is the responsibility of the system to ensure that there are no dangling references. Thus, the representation of relationships in this model is more powerful and convenient than is the relational model.

4.4 Fixed Collection of Types Versus Dynamic Collection of Types

Relational DBMSs have only a single type, called the relation type [18][33]. The definitions of a relation's attributes are the type definition and the rows of a relation are instances of the type. Attributes of relations are non-decomposable and they have simple data types such as integer, real, string, and date. Operations on relations are restricted to retrieving, updating, deleting, and inserting tuples identified by attributes values. New

types are created with a limited set of basic types linearly, but these new types do not allow operations that are different from those defined over relation types. In other words, there is no way to add new operations for the new types.

A fixed set of data structuring primitives will not adequately support design data. Extensions to the built-in primitives, add functionality to a data model at a level indistinguishable from the built-in primitives.

An object-oriented database stores class definitions and objects that are instances of these classes. Class definitions are the same as schemas of the relational systems, but with the important feature that classes encapsulate the behaviour of the object by packaging operations with the data structure [18]. The properties of classes can be simple data types or can be references to other classes of arbitrary complexity. Users can create new classes from existing classes through the facilities of inheritance and tailoring them to the needs of their own applications.

4.5 Data Integrity

4.5.1 Integrity Constraints

In the relational model, expressing integrity constraints with greater semantic content than referential integrity is impossible. For example, expressing the fact that a relationship is one-to-one or one-to-many is not possible. Such constraints must be built into the application code that manipulates the relational database. Since such code is not generally shared among all applications, it is difficult to ensure that data will be updated consistently at all times.

In the object-oriented model, a class defines a data abstraction that has a specification of the operations. The methods or operations can be applied to the instances of the class. A high degree of data independence is achieved by defining the database in terms of such abstractions. This means that it is possible to change implementation of a class without affecting other classes or transactions that use the abstraction. New classes can be created from existing classes by supplying a specification and implementation for new properties and operations. Since these properties and operations are usually implemented by means of a general-purpose programming language, user-defined operations that may include consistency checks for data integrity, can be constructed and stored in the database. Instead of using many database operations for data integrity as is done in the relational systems, an operation on an object can be used. This means that the operation may be used by every transaction. Also, by storing operations as part of the database, a large part of application code can be under the automatic control.

For example, a university database that keeps the students data, has some constraints, for instance, on specific courses. The courses may have prerequisites, and limited enrolment.

In relational systems, the constraints are usually checked by the application codes. Every application code that adds students to the courses, must contain codes to check the constraints. Each time the constraints are changed, all the application codes must be updated.

In object-oriented systems, the database can encapsulate operations that check the constraints with the course objects themselves. This guarantees that every time an application code accesses the courses, the constraints will be satisfied. If the constraints

changes, it is easy to update the database to reflect changes in constraints because the code for constraints is local.

4.5.2 Entity Integrity

In the relational systems, a primary key is an attribute or a minimum collection of attributes whose values uniquely identify an entity. The properties of uniqueness are not always present in the real world. Moreover, it is often necessary to introduce artificial identifiers for entities. Any change to the value of primary keys may cause inconsistency in the system.

In the object-oriented systems, all objects have a unique system-generated identifier which other objects can refer to that identifier. Any change to the value of an object, does not affect its identifier.

4.6 Query Languages

4.6.1 Process of Navigation

Relational systems are intended to perform operations on large sets of data. They perform navigation by using joins. Without using appropriate indices, relational joins are several orders of magnitude slower than pointer traversal.

Object-oriented systems are efficient at quickly navigating from one object to another by traversing pointers. Operations are performed on individual objects.

4.6.2 Query Optimisation

In the relational model, queries are cast in terms of well defined operators over relations. Therefore, translating query expressions into equivalent forms to optimise the queries is an easy task.

In the object-oriented models, queries can involve operators for newly defined abstract data types. Each new type, by introducing new operations, creates a new algebra whose properties are unknown to the query optimiser.

Another problem for query optimisation in object-oriented databases is encapsulation of implementations by abstract data types. Since processing costs are typically dependent on the underlying storage structures for the objects and their aggregates, knowing about the existence of such storage structures seems at first to be a violation of encapsulation [10].

4.7 Application Programs

4.7.1 Database Semantic

In relational systems, the semantic content of the database is scattered among the application codes.

In object-oriented systems, more information can reside in common repository, including information about the semantic content of the database that can be captured in encapsulated methods (operations) with data [37].

4.7.2 Impedance Mismatch

In RDBMSs, there is the problem of impedance mismatch between the programming languages and the database language statements.

In OODBMSs, there is a seamless integration between the programming languages and the database.

4.8 Security

In object-oriented database systems, class definitions provide data abstraction that hide the internal representation of objects (instance variables) from the users of those objects [15].

In relational database systems, hiding of named resources is accomplished through a view mechanism. Views are usually computed as a function, i.e. a query, on some base set of definitions. Execution of the query returns a relation through which the user of the view can access the database.

4.9 Standards

RDBMSs as described in the 1986 ANSI SQL standard, have a standard model for data definition and retrieval. The SQL is of great importance to the success of the relational systems.

OODBMSs, have not an accepted standard data model. Also, there are no guidelines for designing object-oriented databases. The lack of standards has contributed to the lack of useful tools for the user and developer of the system.

4.10 Summary

The basic differences between relational, and object-oriented data modelling were considered. One of the fundamental features of the relational model is the formal specification of the entire data model, where there is no such definition for the object-oriented model.

Object identity is one of the important features of object-oriented DBMSs over relational DBMSs. Each object has a unique system generated identifier that is independent of the attribute values of the object. In RDBMSs, primary keys are user-supplied scalar values which are used to construct an identifier key representing the identity of an object.

Relational DBMSs have only a single type, called relation type. In object-oriented DBMSs, the properties of classes can be simple data types or can be references to other classes of arbitrary complexity.

In the relational system, the constraints are usually checked by the application codes. In object-oriented systems, the database can encapsulate operations that check the constraints. In the relational model translating query expressions into equivalent forms to optimise the queries is an easy task, because queries are cast in terms of well defined operators over relations, but in the object-oriented model, queries can involve operators for newly defined abstract data types.

Chapter

5

Integration of Object-Oriented and Relational DBMSs

In this chapter, a general method, called Database Integration Methodology (DIM), for integration of several heterogeneous database systems, is outlined. Then architecture of a system based on DIM as one particular case of the general method for integration of object-oriented and relational database systems is considered.

To hide heterogeneity of participating DBMSs in a system based on DIM (from now on we call it DIM system), a unified data model and a database language are selected. Schemas of the databases are translated into the unified data model, and database language statements of the DIM system are translated into language statements of underlying database systems. The general method shows how language statements of different database systems can be combined to have a shared access to the databases.

In the particular case of the general method, relational data model and SQL are selected as the unified data model, and the database language statement, respectively. A

solution for translation of object-oriented database schemas to relational schemas is examined.

5.1 Database Integration Methodology (DIM)

Database Integration Methodology (DIM) is a general method for shared access to multiple heterogeneous DBMSs. The database systems are autonomous, i.e. they function independently of each other. They are also connected to each other by a communication system. A system based on DIM is a software package, implemented on the top of the database system. It is an interface to the databases, and it provides the global user with a single data model, and a single database language to uniformly access and modify the databases (Figure 9).

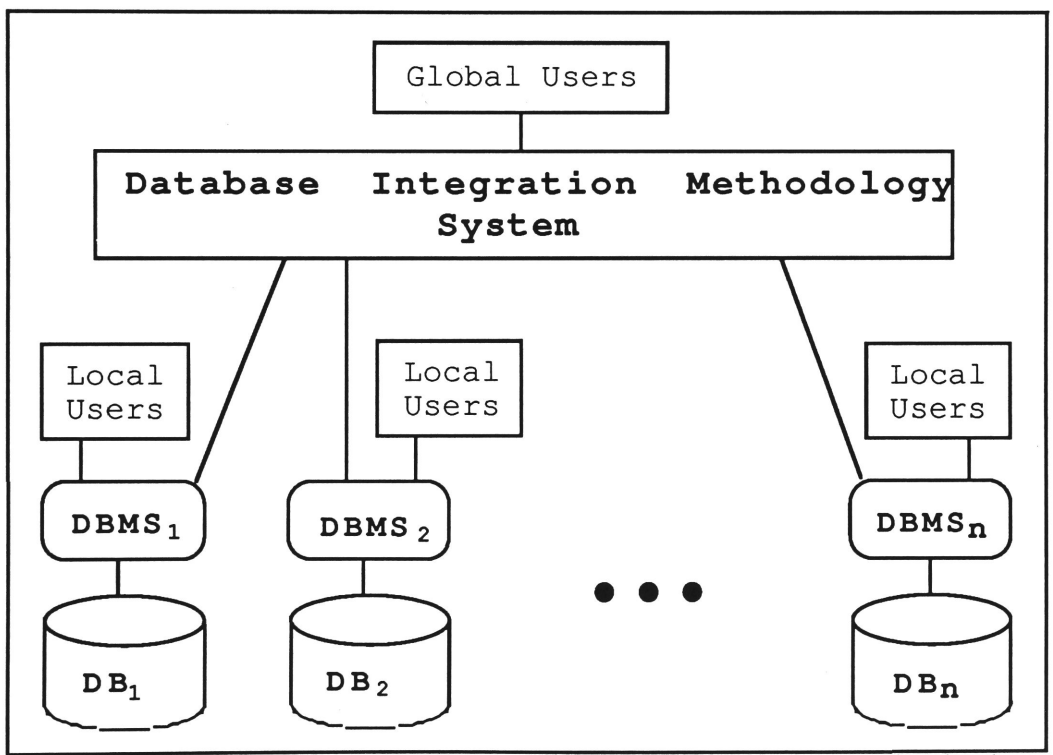

Figure 9. DIM system on the top of DBMSs makes shared access the databases.

One of the significant aspects of a DIM system is that a database system can continue its local operations, and at the same time participate in the DIM system. The integration of database systems is managed by the administrator of the DIM system together with the administrators of the database systems.

The same as a DBMS, a DIM system has a data model through which the global user views the databases as a single database. It has also a query language that is used to uniformly access and modify the databases. This includes hiding the heterogeneity of data models, and database languages of underlying database systems.

Therefore, a data model, and a database language are selected for DIM system. The data model is referred to as a common data model (CDM). It is by means of the CDM that all data models of the DBMSs become as one data model for the user of DIM system. A data model should have some characteristics to be suitable as the CDM [30][34]. For example, it should be rich enough to represent schemas in different data models. Also, in choosing the database language of the selected data model, it is necessary that statements in the language can be translated into statements in languages of other data models. The CDM can be one of data models of the DBMSs, or it can be different from them. However, it is preferable that the CDM be one of the data models of DBMSs because there is no need for the translation of schemas of that database into the CDM.

With selection of a CDM for DIM system, translation of schemas of the participating databases into the CDM is performed (Figure 10). Moreover, schema translation is performed when a schema represented in one data model is different from the CDM. Not all schemas of a database may be available to DIM system and its users. In other words, those necessary schemas that must take part in the DIM system are translated into the CDM, then access to some schemas are granted to the users of DIM system. A

translated schema is in the form of <schema, mapping> which the mapping is a path to the data of the schema [25].

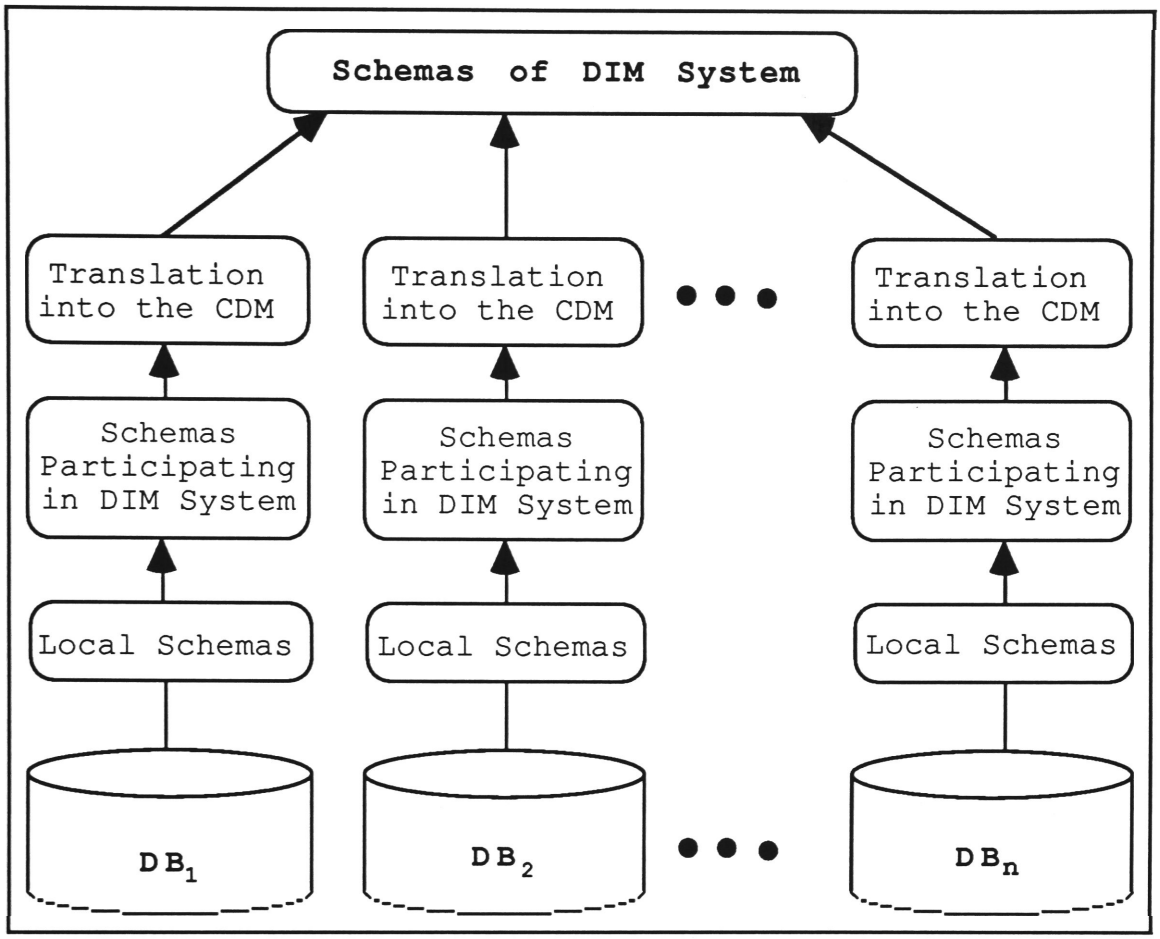

Figure 10. Schemas of DIM system.

Schema integration refers to integrating schemas into a single schema. In DIM system, no schema integration is performed on the translated schemas. Moreover, DIM system assumes that the user needs to access the participating databases without the benefit of a global schema [24].

Query language of a DIM system is favoured in regard to its CDM. When a CDM is one of data models of the databases, it is also preferable to choose the query language of that model. Furthermore, user of DIM system may query or update the database which its

data model is the CDM and its query language has been selected as the language of DIM system. In this case, the query or update statement can directly be given to the DBMS.

When a statement in the language of a DIM system is processed, it is converted to a number of substatements. The substatements are in the form of languages statements of underlying database systems (Figure 11). They are merged togethers as a program in a host language which is common among the host languages of the DBMSs (Section 3.1.7, Host Languages). The substatements must be transformed to calls on procedures provided by the DBMSs. This is performed by the preprocessors. The preprocessors transform the substatements to the database procedures calls of the common host language.

After preprocessing, the program is compiled with the common host language compiler. Then, it is linked with the libraries of the DBMSs to conform an executable code. With execution of the code, the underlying databases are accessed through the database procedures calls. The following example demonstrates how database query language statements of different databases are merged to conform a program in common host language.

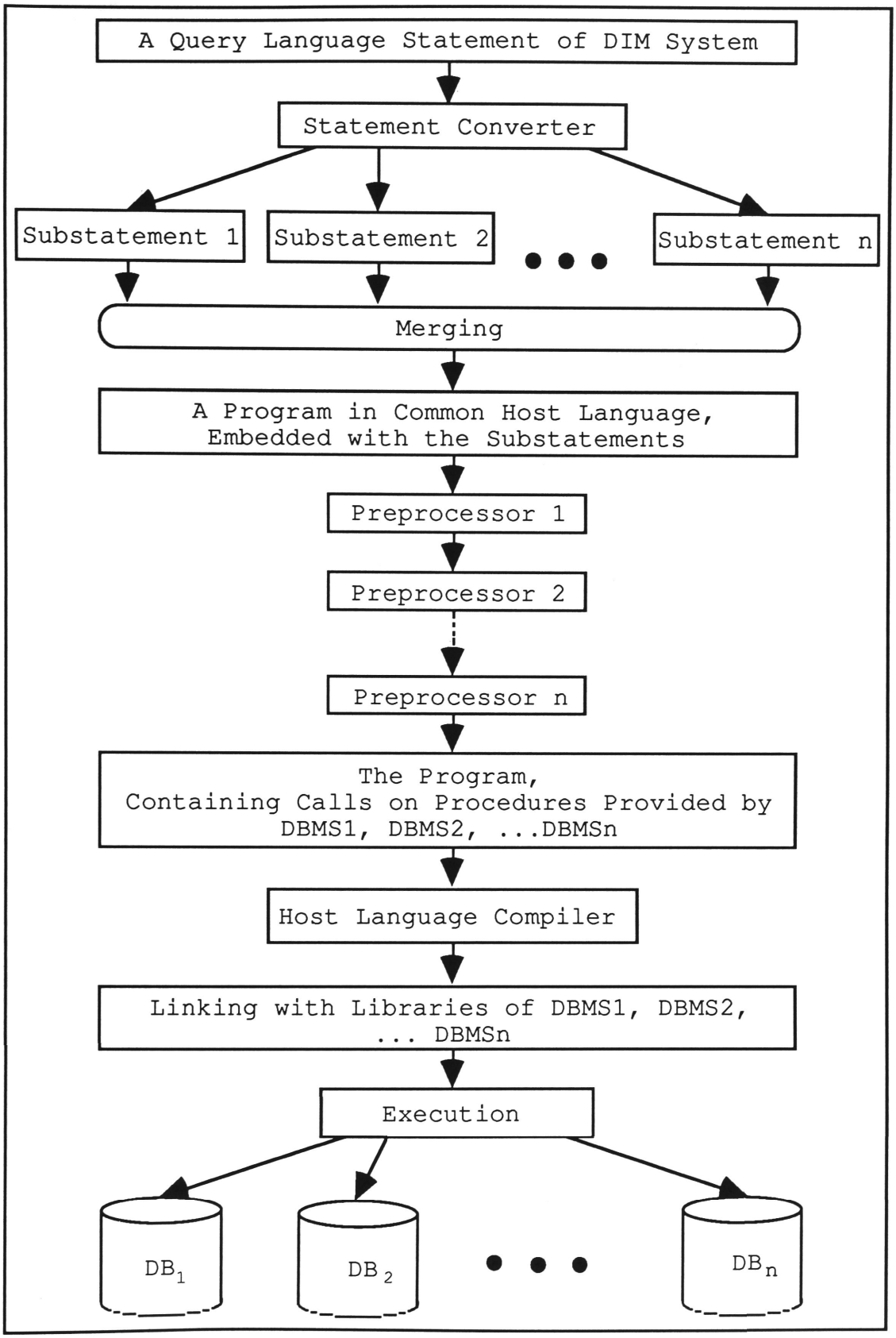

Figure 11. Architecture of a DIM system.

Example: Suppose there are two different DBMSs, called DBMS1, and DBMS2, with the database query languages QUERYLANG1, and QUERYLANG2, respectively. Assume that the following two schemas are the translated schemas of DB1 (belongs to DBMS1), and DB2 (belongs to DBMS2) into the CDM:

DB1:

EMPLOYEE		
EmpNo	EmpName	Address

DB2:

EMPLOYEE_DEPT	
EmpNo	DeptNo

Also, suppose the following statement is a query language statement of DIM system:

```
SELECT  EmpNo, EmpName, DeptNo
        FROM  EMPLOYEE, EMPLOYEE_DEPT
        CONDITION: EMPLOYEE.EmpNo =  EMPLOYEE_DEPT.EmpNo ;
```

This statement is converted to statements of QUERYLANG1, and QUERYLANG2. Then the converted statements are embedded in a host language program. Suppose the converted statements begin with the name of query language, i.e. QUERYLANG1, and QUERYLANG2. A few sentences of the program in common host language is as follows:

```
....
QUERYLANG1  FOR ALL EMPLOYEE RECORDS IN DB1
BEGIN
    QUERYLANG2  FOR ALL EMPLOYEE_DEPT  RECORDS IN DB2
    BEGIN
        IF( EMPLOYEE.EmpNo =  EMPLOYEE_DEPT.EmpNo) THEN
        PRINT(EMPLOYEE.EmpNo, EMPLOYEE.EmpName,
            EMPLOYEE_DEPT.DeptNo ) ;
    END;
END;
```

A common host language has a primary role in the construction of a DIM system. In fact, it is a conventional programming language such as C, PASCAL, COBOL, or PLI which is supported by all DBMSs participating in a DIM system. There may exist many common host languages, supported by underlying DBMSs of which one of them is preferred to be the common host language for DIM system. Also, it may happen that no common host language be found. Therefore, availability of a common host language is critical for constructing a DIM system because it is the common host language that makes the merging of the substatements as a program possible.

Instead of generating one program in the common host language, the alternative approach is to generate multiple programs in different languages, each with embedded queries. The generated programs are preprocessed, and compiled with their preprocessors, and compilers. Then, each links with one DBMS library. This approach has only the benefit of concurrent access to the databases, but has many disadvantages including:

* It is difficult to combine the answers.

* Executions of different transactions must be synchronised.

Therefore, the common host language approach is preferred to different languages.

A data dictionary or data directory plays an important role in coordinating various activities by storing essential information. In addition to storing all the translated schemas, a data dictionary also stores mapping between the translated schemas, and the real

schemas. The mapping includes schema names and their related database names, attribute names of each schema, and access path to data for the attributes.

5.2 Properties of DIM

DIM approach has many properties such as follows:

* It allows operation of existing applications to be continued without any changes.

* It does not impose any reprogramming of database systems.

* It supports controlled integration of existing databases.

* It eases incorporation of new applications and new databases.

5.3 Application of DIM in the Case of Object-Oriented and Relational Database Systems

5.3.1 Common Data Model (CDM) and Query Language

After finding a common host language between the database systems, the next step towards integration of object-oriented and relational systems is to choose a CDM for the DIM system. The CDM represents a unified view of both databases. The use of the CDM solves the problem of syntactic heterogeneities which are consequences of different data

models. Although there are many parameters in the choice of a CDM [30][34], we define two factors in the selection of a CDM for the DIM system as follows:

* The CDM should be easy for the user to learn and to use .
* It should provide the user with the most familiar user's view of data and the most familiar user's query language.

There are many data models such as hierarchical, network, relational, object-oriented, functional, and semantic data models that can be selected as a CDM. Among these data models, the relational and object-oriented data models which are data models of the databases are nominated for the CDM. One reason for this nomination is that if the CDM be one of the data models of the database systems, only the database schemas of the other model is translated into the CDM.

Therefore, the CDM should be relational, or object-oriented data model. It seems that object-oriented data model is preferred to the relational[17]:

* Object-oriented data model is powerful enough to model many new and existing applications.
* Object-oriented data model is as powerful as any other data model; it subsumes all the capabilities of all other data models.

But in [34], it is indicated that "object-oriented data models fails to represent explicitly information of database applications, which in turn causes problems during the integration and the translation steps. Among the limitations of object-oriented data model we cite three critical ones in the context of interoperability. The first one is an inadequacy in representing and manipulating various complex relationships of the database

applications...The second critical problem is a lack of explicit representation and management of the constraints as an element of a database system. Thus the constraints are only partly expressed and their semantics are buried into a method of a class,...The third problem is that the object-oriented data models allow modelling behavior of database applications in procedural language which poses problems for integrating and translating operations expressed in local databases."

Moreover, although object-oriented data modelling is a powerful tool for modelling complex objects and their relationships, it introduces added complexities with which the users must cope [21].

On the other hand, choice of relational data model as a CDM, gives user of DIM system the view of the databases as relational modelling of data. Automatically, SQL which is the standard query language of relational systems will be the query language of DIM system. Choice of the relational model as a CDM relatively satisfies the above mentioned factors. Furthermore, most of the future databases are expected to be provided with the relational interface [24]. Also, there are already many users who are familiar with relational modelling of data. Thus, relational model is selected to be the CDM and SQL the query language of the DIM system.

5.3.2 Comparing Relational and Object-Oriented Terms

Object-oriented systems bring the terms object, class, instance of a class, instance variable, method, object representation, and other object-oriented terminologies. Relational systems introduce relational table, row (tuple), column, flat representation of data, and other relational terms.

In object-oriented database systems, roughly speaking, a class is equivalent to a table, and an instance of the class is equivalent to a row of the table (Table 2). Therefore an instance variable of the class matches to an attribute name of the table in relational concepts.

Object-Oriented Database	Relational Database
Class	Table
Instance Variable	Attribute
Object, Class Instance	Tuple
Instance Variable Constraint	Domain
Message	Procedure or Function Call
Method	Procedure or Function
Collection	Set, Relation

Table 2. An approximate equivalent terms of object-oriented and relational database.

5.3.3 Translation

A person who is called global database administrator (GDBA), or database integrator (DBI) is responsible for the translation of the databases into the CDM. The GDBA is also responsible for the overall consistency of the DIM SYSTEM and storage of the essential information that is required for accessing the databases.

The CDM is relational data model, so the schemas are translated into relational modelling of data. This implies that schemas of the databases are translated into relational

schemas. For schemas of the relational database no translation is needed into the CDM (Figure 12).

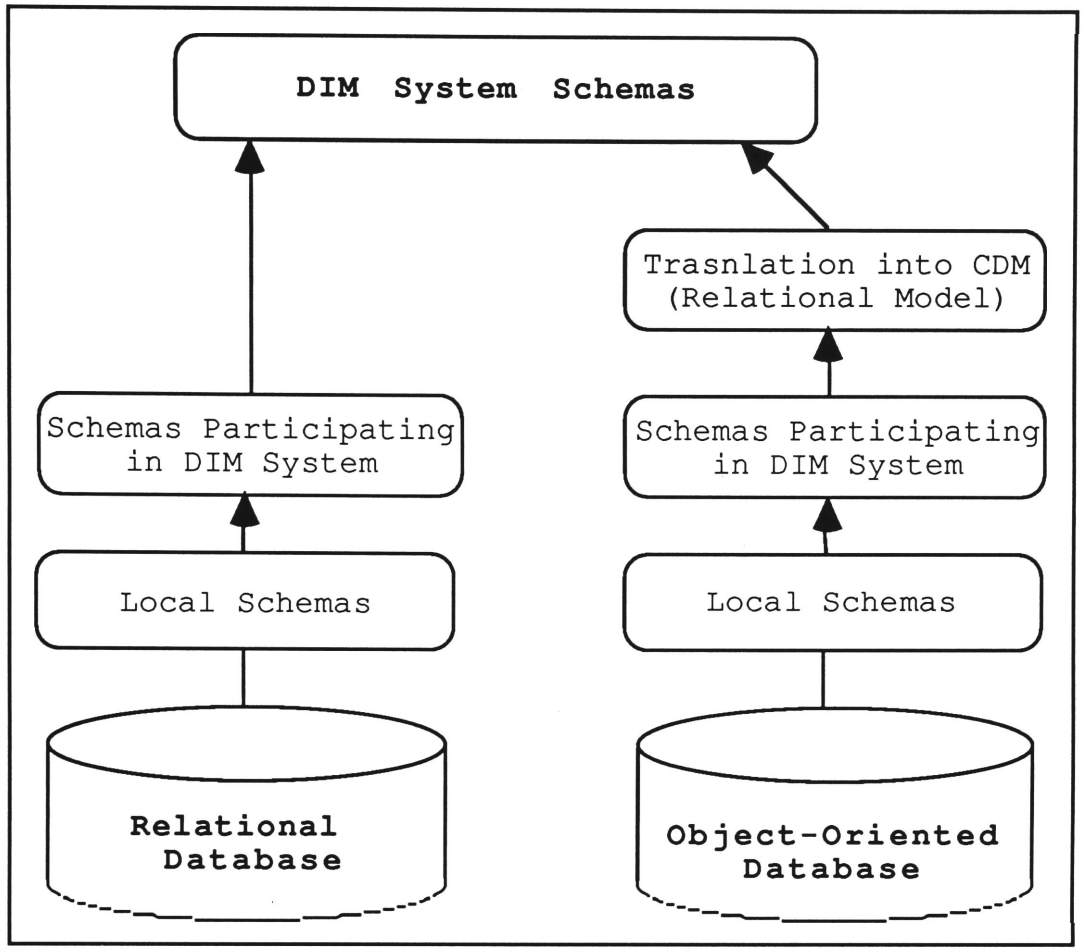

Figure 12. Schemas of DIM system in the case of participating object-oriented and relational databases, and the CDM be the relational model.

In translation of object-oriented schema into relational schema, a class name can be translated into a table name, and an instance variable of the class into a column name of the table. It is not necessary that the names of public instance variables of a class be exactly the same as column names. It is the policy of the GDBA to invent new names or to use the same names.

If objects must act as relational tables, objects structures are decomposed and flattened [29]. Moreover, there may be a complex object, i.e. an object with hierarchical structures. In this case, the complex object is completely decomposed and flatten by

which a relational schema to be constructed. Attributes of the relational schema corresponds with all instance variables in the object. The following example shows how the translation is performed.

Example 1. Suppose an object-oriented database schema, expressed in a pseudo code as follows, and we want to construct a relational schema, based on the schema.

```
class  employee {

  private:
   int     Emp_No ;
   char    Emp_Name[20] ;

  public:
   department *Emp_Dept ;
   address    *Emp_Address ;

  methods:
   create: employee(...) ;
   GetEmpNo: return Emp_No ;
   GetEmpName: return Emp_Name;
   ...

  } ;
```

```
class department {

  private:
    float  Budget ;

  public :
    char       Dept_Name[20] ;
    Projects   *Dept_Project ;

  methods:
    create: department (...) ;
    Get_Budget: return Budget ;
      ...

   };
```

```
class  address {

  public:
    int    Home_No;
    char   Street_Name[20];
    char   Suburb_Name[20];
    char   Zip_Code[10] ;

  methods:
    create: address (...)
;
        ...

 } ;
```

```
class projects  {

  private:
   int     Project_No ;

  public:
   char    Project_Name[20] ;
   char    Date_Started[8] ;

  methods:
   create: projects(...) ;
   GetProjectNo: Return Project_No ;
       ...

};
```

The employee class has references to two classes: the address class, and the department class. The department class has also a reference to the project class The equivalent relational schema after translation of the employee class is represented in Figure 13.

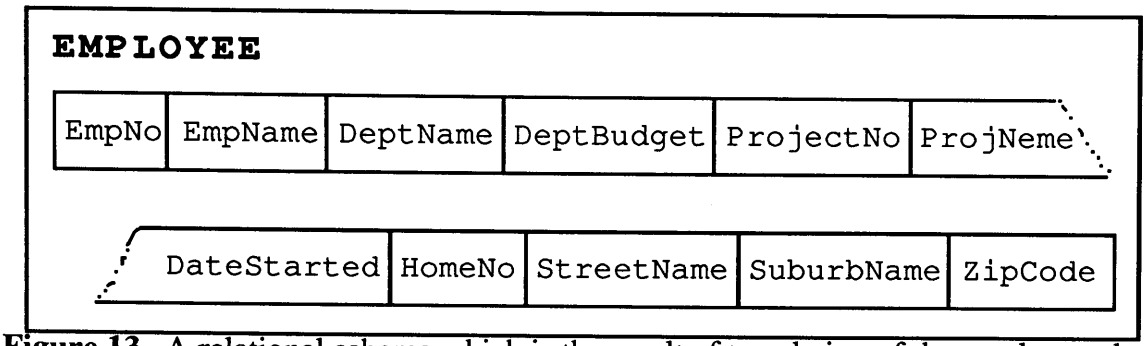

Figure 13. A relational schema which is the result of translation of the employee class in example 1.

For representation of a translated schema in the DIM system, a relational table based on the schema is created. The relational table is stored as an empty table. This table is called virtual table.

Moreover, in the DIM system, there are two kinds of tables: real tables, and virtual tables. Real tables are related to the relational database, and virtual tables to the object-oriented database. The kind of tables, i.e. real and virtual, are hidden from the user. They are used irrespective to that they are real, or virtual tables.

Representation of object-oriented schemas as the virtual tables give the users of the DIM system relational view of the object-oriented database. In fact, the users of the DIM system view the underlying databases as a set of relational tables.

For each column of virtual tables the following information, called virtual tables information (VTI), is stored in the DIM system:

Column name: The name of a column of a virtual table.

Table name: The name of the virtual table.

Class name: The name of the class by which the virtual table is constructed.

Access path: The related path for accessing value from object-oriented database for the column.

In the case of Example 1, the VTI is shown in Table 3.

Table Name	Column Name	Class Name	Acess-Path
EMPLOYEE	EmpNo	employee	GetEmpNo()
EMPLOYEE	EmpName	employee	GetEmpName()
EMPLOYEE	DeptName	department	Emp_Dept->Dept_Name
EMPLOYEE	DeptBudget	department	Emp_Dept->GetBudget()
EMPLOYEE	ProjectNo	projects	Dept_Project->GetProjectNo
EMPLOYEE	ProjectName	Projects	Dept_Project->Project_Name
EMPLOYEE	DateStarted	Projects	Dept_Project->Date_Started
EMPLOYEE	HomeNo	address	Emp_Address->Home_No
EMPLOYEE	StreetName	address	Emp_Address->Street_Name
EMPLOYEE	SuburbName	address	Emp_Address->Suburb_Name
EMPLOYEE	ZipCode	address	Emp_Address->Zip_Code

Table 3. The virtual tables information (VTI) for the translated schema of Example 1.

When a column of a virtual table is used in a SQL statement, the DIM SYSTEM searches the VTI for the pertinent class name and access path to the column. The class name along with the access path make the access to the data in the object-oriented database possible.

5.3.4 Processing a SQL Statement in the DIM System

Virtual and real tables are used in a SQL statement to uniformly access and modify object-oriented and relational databases. In the DIM system, when real and virtual tables are included in a SQL statement, the statement is converted into two separated substatements. One of the substatements comprises the real tables, and is in the form of a SQL statement. The other substatement, which is concerned with the virtual tables, is in the structure of the query language of object-oriented database. The substatements are merged togethers as a program in a common host language of the database systems. The embedded databases substatements in the program are transformed to database procedures calls by the relevant preprocessors. The preprocessed program is compiled with the common host language compiler. Then, it is linked with the libraries of RDBMS, and OODBMS to conform an executable code With the execution of the code, both databases are accessed uniformly (Figure 14).

It is possible to use one kind of the tables in a SQL statement. For example, a SQL statement may either contain virtual tables or real tables. When a SQL statement only contains real tables, and the statement does not have any specific output, the statement is directly given to the relational DBMS. Example of such a statement is to use DELETE statement for a real table.

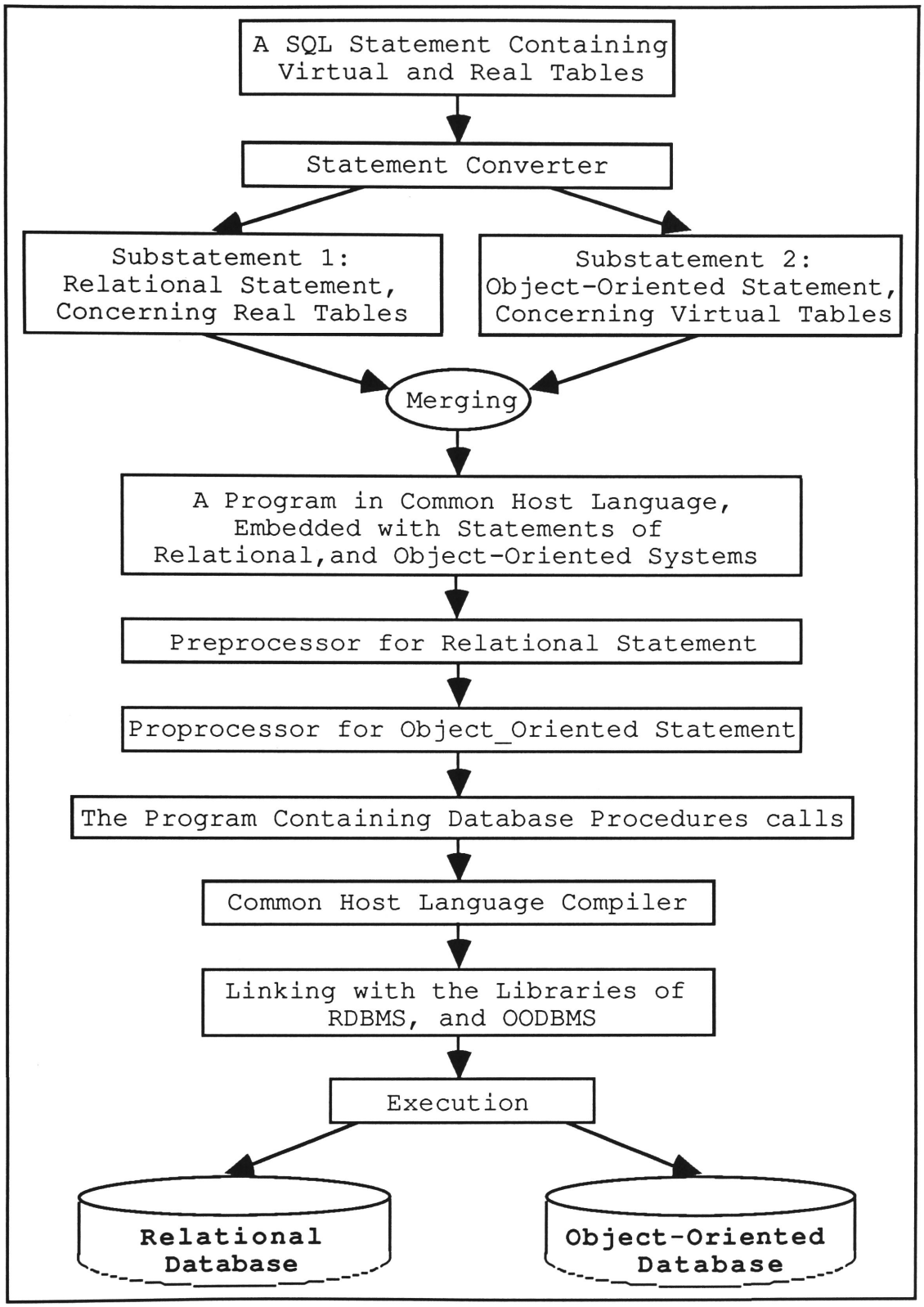

Figure 14. Structure of the DIM system for processing a SQL statement in the case of object-oriented and relational databases.

5.3.5 Different Approaches for Accessing Virtual Tables

To retrieve data for the virtual tables that have been used in a SQL statement, the following two approaches are considered:

1. After a statement is issued, with respecting that which virtual tables are referred in the statement, the appropriate data from object oriented database is accessed.

2. Depending on which virtual tables in a SQL statement are used, the DIM system creates a table from join of the virtual tables, and inserts the related data from object-oriented database into it. Immediately, this table is used as a real table in the DIM system, and if it is stored in the relational database, the statement with a little change can be given to the relational DBMS for processing.

In the first approach, when a SQL statement includes virtual tables, the virtual tables information (VTI) is searched for the appropriate classes names and access paths for the virtual tables. With the classes names and access paths, the object-oriented database is accessed. Instances of the classes construct rows of the virtual tables. The necessary data manipulation concerning to the statement is performed on the instances of the classes for the desired output by the DIM system.

In the second approach, access to the object-oriented database is performed the same as the first approach but a new table form join of the virtual tables is created and the data from the object-oriented database is inserted into the table. With storage of the table in the relational database, processing of the SQL statement is carried out by the relational DBMS.

Despite that the second approach provides easier query processing than the first one, it increases more overheads on the DIM system. Moreover, in the second approach, the time for creating a relational table and inserting data from the object-oriented database into this table is added to the time of processing a query in the DIM system.

Example 2. Suppose the EMPLOYEE and PART_RESPONSIBLE schemas (Figure 15) are the translated schemas of the object-oriented database, and the PART schema is a schema which belongs to the relational database (the PART_RESPONSIBLE schema is a part of translated schema of a class that one of its reference attribute is the employee class, and translation of the employee class is the EMPLOYEE schema).

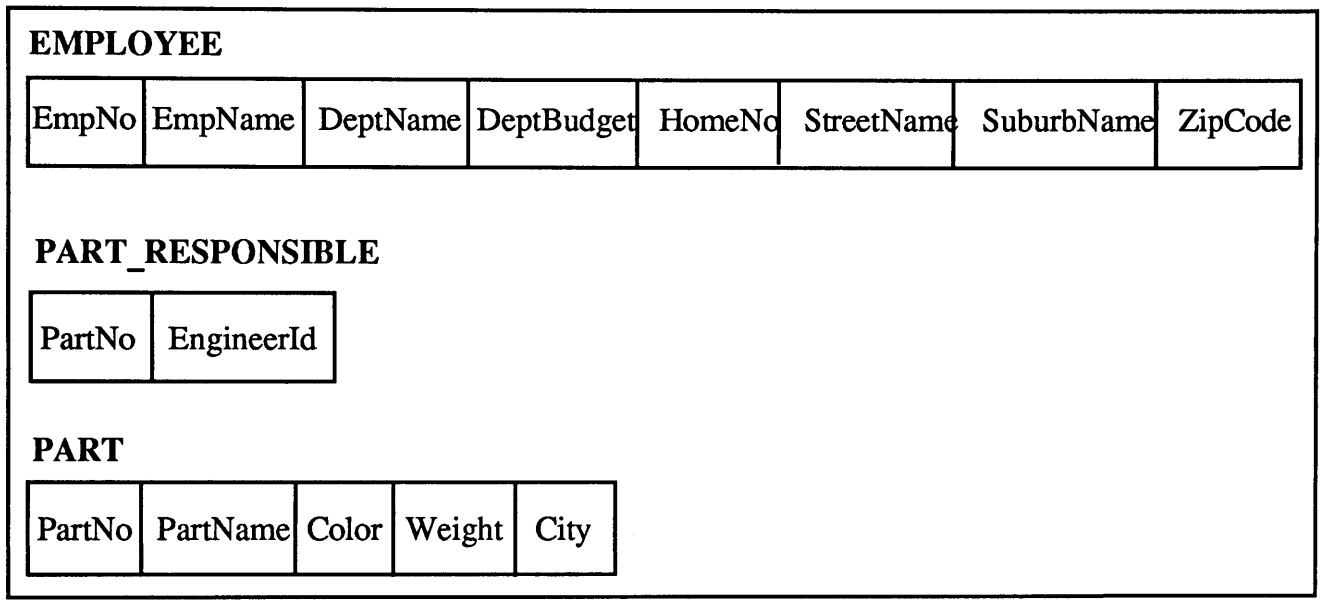

Figure 15. The EMPLOYEE and PART_RESPONSIBLE are translated schemas of the object-oriented database, and the PART schema is a schema of the relational database.

The following query is a SELECT statement that uses two virtual tables: EMPLOYEE, PART_RESPONSIBLE, and one real table: PART.

```
SELECT   EmpName, DeptName, PartName
    FROM        PART, EMPLOYEE, PART_RESPONSIBLE
    WHERE       PART_RESPONSIBLE.PartNo = PART.PartNo
    AND         EngineerID = EmpNo ;
```

As the result of execution of the SELECT statement, a new table is created from the join of the virtual tables EMPLOYEE, and PART_RESPONSIBLE, then the related data from object-oriented database is inserted into the table. Storage of the table in the relational database, makes processing of the query easier.

Suppose the created table is called EMPLOYEE_PART_RESPONSIBLE. The SELECT statement can be changed to a statement so that it can be directly submitted to the relational DBMS. To change the SELECT statement, the virtual tables EMPLOYEE and PART_RESPONSIBLE of the SELECT statement are replaced with the created table EMPLOYEE_PART_RESPONSIBLE as follows:

```
SELECT   EmpName, DeptName, PartName
    FROM
        PART, EMPLOYEE_PART_RESPONSIBLE
    WHERE
        EMPLOYEE_PART_RESPONSIBLE.PartNo = PART.PartNo
    AND
        EngineerID = EmpNo ;
```

Processing of the new statement is left to the relational DBMS. This makes processing of the query easier than processing of it by the DIM system.

It is also possible to optimise the table, when creating it, and when inserting the relevant data into it. Moreover, the created table can be a table which its columns are restricted to the columns that are used in the query, and its rows can be inserted in regards to the condition in that query. For example, the new table can be created with EmpName, DeptName, PartName, PartNo, EngineerID, and EmpNo as its columns. Further, with employing condition of the SELECT statement (i.e. EngineerID = EmpNo) when the data from object-oriented database is inserted into the table, the number of inserted rows into the table may be decreased. In fact, the new table with applying the condition of the query can be created with fewer columns and possibly fewer rows. For example, the columns can be restricted to: EmpName, DeptName, PartName, and PartNo.

Despite many advantages of the second approach, it has several side effects on the DIM system. The side effects are:

* The time for creating a new table and inserting the related values into that table increases the response time for processing a SQL statement in the DIM system.

* Storage of the table in the DIM system means to copy a part of a database into another place.

* If a statement updates the table, the updates must be immediately reflected to the object-oriented database.

* When the process of a SQL statement is finished, preserving such a table in the DIM system is useless. Further, the object-oriented database may be updated by local users, but the table still contains the old values. Also, it is not reasonable to

maintain such tables in the DIM system, perhaps the same statement be used in future.

* It violates the autonomy of the relational database.

Because of the foregoing side effects, the second approach is completely unsuitable for use in the DIM system. Instead, in the first approach, straight retrieval and update of the object-oriented database is preferred. In comparison to the second approach, the first approach removes the problem of inconsistencies and redundancies of data that might occur for object-oriented database. Therefore, the first approach for accessing and updating of the object-oriented database is selected.

5.3.6 Virtual Tables and Normalisation

In the DIM system, virtual tables are means for the user to perceive the object-oriented database as a relational database, and to query them. Queries against virtual tables transform to queries of the object-oriented DBMS. Further, when a SQL statement containing virtual tables is processed, rows of the virtual tables are constructed in the memory of the system. The rows are instances of the related classes that have been accessed from the objected-oriented database.

Columns of virtual tables represent atomic values. Therefore, virtual tables are normalised relations, and they are in first normal form. Since representation of virtual tables has not any value, i.e. they are empty relational tables, they do not need any further normalisation.

5.3.7 Responsibilities of the Global Database Administrator (GDBA)

The DIM system is the type of a loosely coupled HDBMS, therefore the GDBA has many responsibilities for accommodating the desired operation of the system [5][32]. In the previous sections, some of the responsibilities of the GDBA have been discussed. In this section, they are considered in more detail.

The GDBA with cooperation of the local DBAs study which schemas of underlying databases should participate in the DIM system. For the relational database, for example, the GDBA establishes the appropriate authority for participating the relational tables in the DIM system. In the case of object-oriented database, the GDBA translates the participating schemas of the object-oriented database into their equivalent relational schemas, i.e. the virtual tables. The work of the GDBA is concentrated on the schema translation, and resolving semantic heterogeneity among different schemas. Cooperation of the GDBA with local DBAs for resolving the semantic heterogeneity, is inevitable.

Essential information about schemas is stored in the DIM system. After translation of the schemas, the GDBA creates the virtual tables in relation to the schemas. The GDBA also constructs the virtual tables information (VTI) for each column of the virtual tables, and stores them in the DIM system.

5.3 Summary

A Database Integration Methodology (DIM) is a general method for making shared access to heterogeneous databases through a common host language. Finding the common

host language among the participating database systems in constructing a system based on DIM (DIM system) is vital. Without the common host language, shared access to the databases is not easily feasible in DIM system.

To hide the heterogeneity of the data models, and the database languages of the underlying databases, a DIM system has a data model called Common Data Model (CDM) through which the user views the databases as a single database, and it has also a query language that is used to concurrently access all the databases.

For the case of object-oriented and relational database systems participating in a DIM system, the CDM was chosen to be the relational modelling of data. Simplicity and prevalence of the relational model were the major factors in choosing it as the CDM for the DIM system.

In the DIM system, virtual tables are means for the user to view the object-oriented database as a relational database, and to make query and update the tables. The necessary information about virtual tables are kept in the virtual tables information (VTI). For each column of virtual tables, the VTI contains the pertinent information to the column for accessing the object-oriented database

The next chapter outlines the implementation of a DIM system in the case of object-oriented and relational database systems.

Chapter

6

Implementation

In this chapter, implementation of a system based on DIM for integration of object-oriented and relational database systems is considered.

6.1 The Environment and Possibilities

For the implementation of this project, ORACLE, and ObjectStore were used. ORACLE is a relational DBMS, and ObjectStore is an object-oriented DBMS. In addition to the interactive interface of SQL statements, ORACLE supports embedded SQL in a programming language such as C which is called Pro*C. For query language, ObjectStore uses a notation based on an algebra derived from C++ expressions.

First step toward the implementation was to find a common host language. If the relational system supported embedded SQL in C++ language, a Pro*C++ (i.e. embedded SQL statements in a C++ program) could be easily constructed, and the problem of the

common host language would be solved. The possibility of using the ObjectStore C++ language was discovered to be a suitable common host language. It means to use embedded SQL statements in a program written in an ObjectStore C++ language.

In the relational system, the C preprocessor is used for preprocessing Pro*C programs. It checks syntax and some semantics of embedded SQL statements in the C programs, and generates some data structures and the database function calls in C for the statements. The resulted output of a preprocessed Pro*C, is a C program in which the embedded SQL statements have been replaced with the data structures and the database function calls. The preprocessed Pro*C code must be compiled with a C compiler, and linked with the related database libraries before it can be executed. Therefore, the C preprocessor, only changes the embedded SQL statements of a Pro*C.

After preprocessing a Pro*C++ program (i.e., an ObjectStore C++ program embedded with SQL statements), the program is transformed so that it can be compiled with the ObjectStore C++ compiler. The program is then linked with the libraries of ObjectStore and ORACLE for execution. Transforming a preprocessed Pro*C++ program is not a difficult and time consuming task. It is performed by a tiny program.

6.2 Processing SQL Statements

With the selection of CDM as relational model, the virtual tables are stored as empty tables in the relational database. The virtual tables information (VTI) is also stored as a relational table in the relational database. The virtual tables are related to the object-oriented database, and the real tables are related to the relational database. The user enters SQL statements to uniformly access the databases.

For any SQL statement which the user enters, the following actions are performed (Figure 16):

1. The statement is checked for the right syntax and semantics.

2. If the statement includes columns of tables, it relates columns to its pertinent tables by accessing the data dictionary of the relational system.

3. It determines which one of the tables in the statement are virtual tables and which one are real ones.

4. It recognises the kind of the SQL statement for interpreting it, then generates an ObjectStore C++, Pro*C, or Pro*C++ program depending on whether the statement includes the virtual tables, the real tables, or both kinds of tables, respectively.

5. If it is necessary, the generated program is preprocessed and transformed. Then it is compiled, and linked with the library of ORACLE, ObjectStore, or both libraries for execution.

To check the syntax of a SQL statement, recognise the type and interpret that statement; to generate a program which checks the semantics of the statement, and to generate another program to access the databases for the desired output, the YACC has been used.

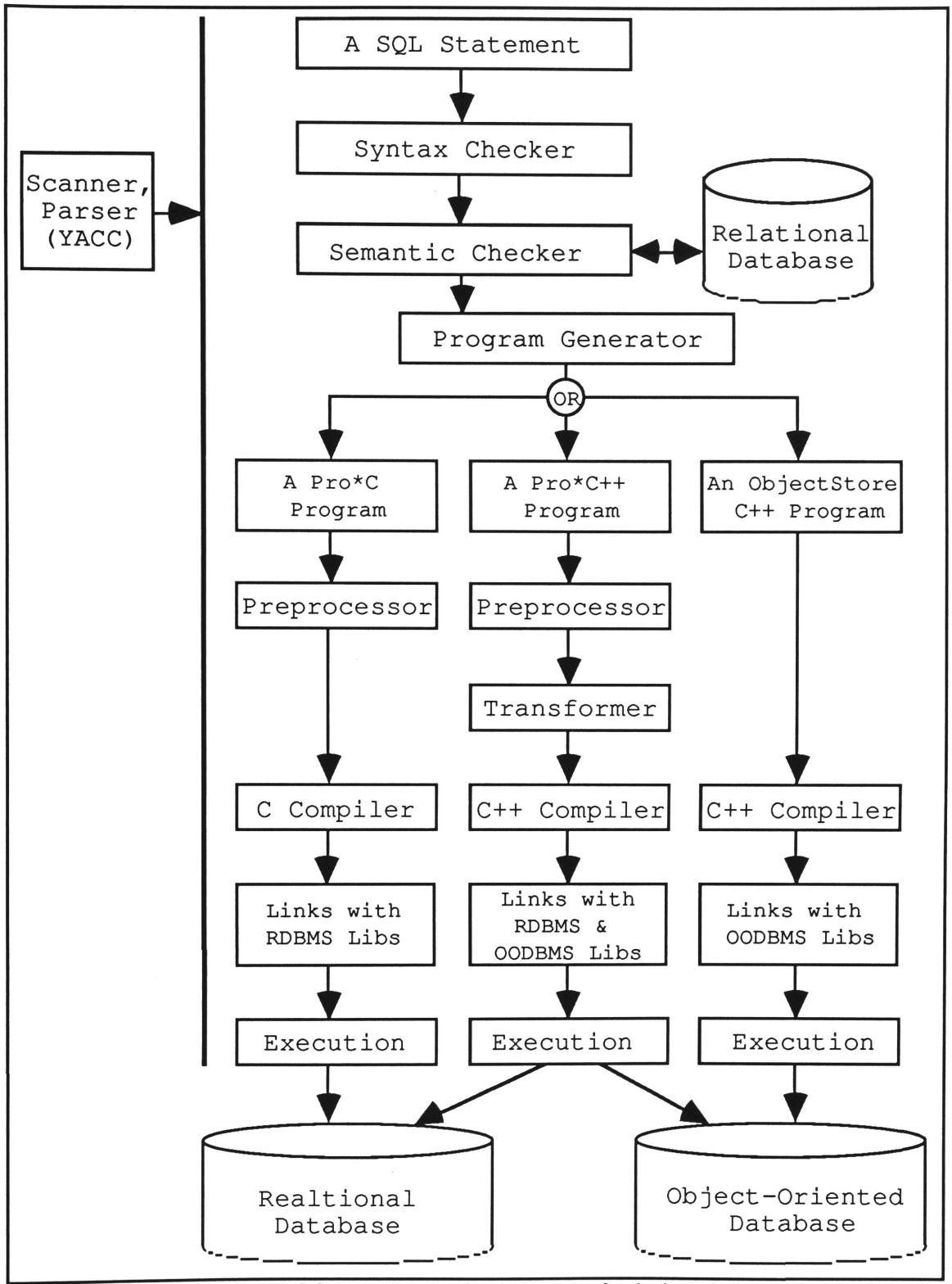

Figure 16. Processing a SQL statement in the case of relational and object-oriented databases in the DIM system.

A scanner, and a parser is needed in order to scan and parse SQL statements to generate a Pro*C++ , Pro*C, or an ObjectStore C++ program. The scanner scans a SQL statement as input and passes each token to the parser. For the parser, the YACC parses SQL statements. The YACC provides a general tool for imposing structure on the input. For each SQL statement, a rule describes the input structure and an action to be invoked when the rule is recognised.

The implementation is slightly different with what has been proposed in chapter 5. This is because the ObjectStore C++ is used as the common host language, and the generated program is compiled with the ObjectStore C++ compiler. Thus, it is necessary to use only one preprocessor instead of two preprocessors.

6.2.1 A SQL Statement

In the DIM system, the user enters a SQL statement interactively. The statement must be ended with a semicolon as a statement terminator, and a "return" key. The user can enter many "return" keys before the semicolon.

SQL statements are used in the same way as their standards. A SQL statement may include virtual tables, real ones, or both kinds of tables. For the user, there is no difference between the real and virtual tables.

6.2.2 Syntax Checker

Syntax of SQL statements are defined as a set of rules in a YACC program. When the user enters a SQL statement, it is scanned, and tokens from the scanner are passed to

the YACC program. Based on the rules for SQL statements, if the syntax of an input SQL statement is wrong, the program will issue the message "Syntax Error".

6.2.3 Semantic Checker

When a SQL statement is verified for syntax, it will also be checked for the correct semantics (i.e. existence of table names, and column names, used in the statement). In the YACC program, the table names and column names of a SQL statement are separated. Since all tables of the DIM system (real, and virtual) are stored in the relational database, by referring to the relational database, the semantics of the statement are checked for correct table names and correct column names. Also, the data dictionary of the relational system is used to relate each table to its columns which have been used in the statement.

6.2.4 Program Generator

A Program is generated when the right syntax, and semantics of a SQL statement have been verified. Depending on the SQL statement, that includes real tables, virtual, or both tables, a Pro*C, an ObjectStore C++ program, or an ObjectStore C++ program embedded with appropriate SQL statements (Pro*C++) is generated. Furthermore, virtual tables and their columns in the SQL statement are converted to ObjectStore statements. Real tables and their columns in the SQL statement are converted to a set of SQL statements. The ObjectStore and the SQL statements are merged to make a Pro*C++ program.

6.2.5 Preprocessor

A generated Pro*C++, or Pro*C program is given to a preprocessor for replacing the SQL statements with database function calls. The preprocessor is normally used for a Pro*C to generate a program that can be compiled with a C compiler. In fact, the preprocessor replaces only embedded SQL statements in the program with some data structures and database function calls. If the generated program is Pro*C, it is passed through the preprocessor. The preprocessed Pro*C program is compiled with C compiler, and it accesses only the relational database. If the generated program is a Pro*C++ program, in addition to preprocessing, it must be transformed such that to be compiled with the C++ compiler. The compiled program after linking with the databases libraries accesses both databases.

6.2.6 Transformer

The preprocessed Pro*C++ program must be transformed such that it can be compiled with the C++ compiler. With preprocessing, embedded SQL statements in a Pro*C++ program are replaced with C language statements. Therefore, the preprocessed Pro*C++ program consists of a number of C language statements, and a number of C++ language statements. Since the program is compiled with the C++ compiler, the C language statements must be changed to the statements acceptable to the C++ compiler. A transformer which is a tiny program does this job. Adding prototypes to the database function calls, is an example of the transformer.

6.3 SQL Statements

The SQL statements that have been implemented are:

SELECT

UPDATE

INSERT,

DELETE

To show some examples about the SQL statements, a set of sample schemas related to object-oriented, and relational databases are used (Figure 17). Also, the VTI related to one object-oriented schema is depicted in Table 4.

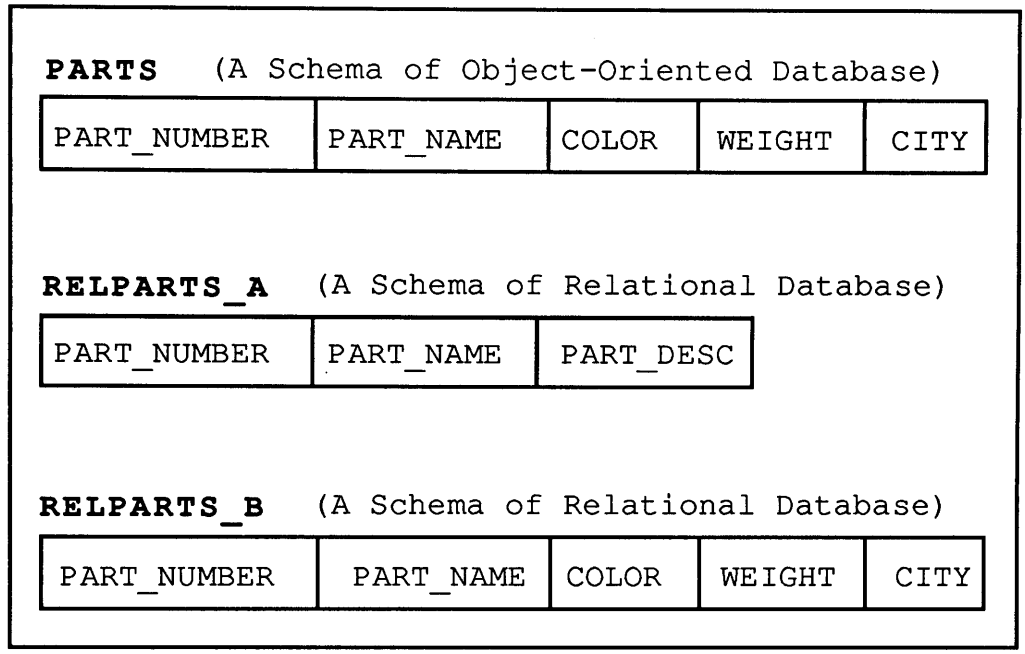

Figure 17. The schemas that are used for the SQL examples.

There are three schemas in Figure 17. One schema is related to object-oriented database, and is stored as an empty table in the relational database, called the virtual table PARTS. The other schemas belong to the relational database, and one of them called RELPARTS_B is exactly the same as the PARTS schema of object-oriented database. Choice of two similar schemas is intentionally to show replicated data in both databases, and to simply insert data from one database to another.

Table Name	Column Name	Class Name	Acess-Path
PARTS	PART_NUMBER	PARTS	GetPartNo()
PARTS	PART_NAME	PARTS	part_name
PARTS	COLOR	PARTS	color
PARTS	WEIGHT	PARTS	GetWeight()
PARTS	CITY	PARTS	city

TABLE 4. A part of the virtual tables information (VTI) related to the PARTS schema.

SELECT. Initially, One level of the SQL SELECT statement was implemented. Implementation of a nested SELECT statement needs more interpretation to be performed on the statement. A nested SELECT statement is a SELECT statement that contains one or more SELECT statements in its condition part. The condition for the implemented SELECT statement can be any conditional expressions such as BETWEEN, IN, NOT IN, and comparison of numeric and string values.

Example (1) SELECT:

```
SELECT  PARTS.PART_NUMBER, COLOR, WEIGHT, PART_DESC
        FROM
        PARTS, RELPARTS_A
        WHERE
        PARTS.PART_NUMBER = RELPARTS_A.PART_NUMBER ;
```

The resulted table is a table from joining the tables PARTS, and RELPARTS_A that holds the specified condition, and has the columns used in the statement.

Example (2) SELECT:

```
SELECT  *
        FROM  PARTS
        WHERE  PART_NUMBER = 'p5' ;
```

The SELECT of Example (2) is a selection of all columns of the virtual table PARTS. The generated Pro*C++ programs as the results of execution of the SELECT statements (Examples 1, and 2) are given in APPENDIX A.

UPDATE. With UPDATE statement, one or more columns of one table (virtual or real) can be updated by using literal values.

Example. UPDATE:

```
UPDATE PARTS
        SET  COLOR = 'red' ,  WEIGHT  =  20
        WHERE  PART_NUMBER = 'p1' ;
```

The UPDATE statement of Example 3, updates two columns of the virtual table PARTS that leads to the updates of instance(s) of the class PARTS of object-oriented database if the specified condition holds.

INSERT. INSERT statement is used to insert a row in a table. For virtual tables, the insertion of a row with literal values, or the retrieved values from relational tables is transformed to creation of an object in the object-oriented database.

Example (1) INSERT:

Insertion of a row in the virtual table with literal values:

```
INSERT INTO PARTS
        VALUES ( 'p2' , 'bolt' , 'yellow' ,  10, 'Wollongong' ) ;
```

A row or a set of rows can be inserted into a virtual table by retrieving the relevant values from another table(s):

Example (2) INSERT:

```
INSERT INTO PARTS
    SELECT   PART_NUMBER, PART_NAME,
                COLOR, WEIGHT, CITY
    FROM   RELPARTS_B
    WHERE   CITY = 'Wollongong' ;
```

Also, a row can be inserted into a relational table with respect to some values retrieved from object-oriented database.

DELETE. With DELETE statement, a row or a set of rows is deleted from tables in regard to the condition of the DELETE statement. Deletion of a row from a virtual table leads to deletion of the related object in the object-oriented database.

Example. DELETE:

```
DELETE FROM PARTS
    WHERE PART_NAME = 'bolt' ;
```

6.4 View Creation

In a relational database system, when a view is created, the definition of that view in terms of other tables is stored in the catalog. In the DIM system, it is possible to create a view from underlying virtual tables with a mechanism that when a view is created, the related information about the view to be stored in the system as a program. Consequently, when the view is referred, the program about the view will be activated.

```
CREATE VIEW
    AS SELECT
        PART_NUMBER, PART_NAME, COLOR
        FROM  PARTS
        WHERE  WEIGHT < 50 ;
```

6.5 Arrays

Usage of composite structures such as arrays are usual in object-oriented systems. In the DIM system, access to arrays is necessary because in some objects template(classes), arrays may be used.

In relational systems, a table has generally a fixed number of columns. For example, a table can be created with the limited number of 256 columns. If there is an array with the number of elements greater than the number of columns limitation, the array elements can not be stored in one table. One way for storing all the array elements as a related elements is to split the array, and store each part in a table. To relate the pertinent element values to each other, artificial foreign keys and primary keys must be invented.

To reveal arrays to the user in the DIM system by the means of virtual tables, some conventional symbols are used. An array constitutes one column of a virtual table. The column name of the virtual table is represented by using a name for that array, an underscore, and the number of array elements. Nested structures are demonstrated by structure names followed the symbol "@" instead of the symbol ".". A structure containing arrays illustrated in Figure 18, consists of a one-dimensional array of 100 elements, and another structure which embodies two one-dimensional arrays of 80, and 50 elements.

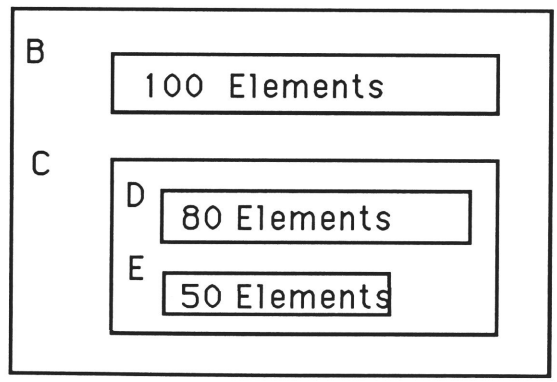

Figure 18. A structure containing one-dimensional arrays.

A virtual table in relation to the structure of Figure 18 can be created as followings:

```
CREATE TABLE Printed_Board (
```

```
B_100        NUMBER(10,2),
C@D_80       CHAR(10),
C@E_50       NUMBER(8) );
```

The user can refer to each element of the arrays by specifying the array name and the element number of that array inside a pair of parenthesis. For example, the fifth element of array "B" is referred as B(5), and the twentieth element of array D inside the structure C is referred as C@D(20). The following SQL statement shows the usage of arrays in a query.

```
SELECT B(85)
        FROM Printed_Board
        WHERE C@D(15) > C@E(30);
```

Multi-dimensional arrays can be created and referred in regards to the dimensions. For example, a two-dimensional array such as "E" with 20 by 30 elements can be created in a CREATE statement as "E_20x30 NUMBER(5),". To refer to the twelfth element of "E", the notation E(3,4) can be used. The symbols are conventional, and for different systems another accepted symbols may be used.

6.6 Summary

The implementation of a DIM system in the case of object-oriented and relational databases was discussed. The main problem for the DIM system was to find a common host language between the database systems. The ObjectStore C++ was discovered to be a suitable common host language.

After translation into the CDM, the participating schemas of object-oriented database in the DIM system are stored as empty tables in the relational database. These tables are called virtual tables but from the point of view of the user they are as real tables in the DIM system. With this mechanism, the user views the databases as a set of relational tables.

The virtual tables information (VTI) which is information about the virtual tables is also stored as a table in the relational database. The VTI is the means for the DIM system to find the access path for each column of virtual tables to access the object-oriented database.

Chapter

7

Conclusions

The lack of standards in HDBMSs directs most vendors and end users to invest separate solutions for their problems. It is believed that HDBSs are not mature and require more extensive research. Therefore, at this point, standards in the area of HDBMSs can not be fully defined or enforced.

In this thesis, we introduced a general method, called Database Integration Methodology (DIM), for integration of several heterogeneous database systems. Preservation of local autonomies of the integrated databases allows for utilisation of the already existing applications without any additional modifications.

A system based on DIM, called DIM system, primarily solves the problem of syntactic heterogeneity of the databases through a CDM which is a unified data model of underlying database systems. DIM system has also a single query language to uniformly access the databases. Therefore, the user views the databases as one database in the CDM. Schemas of participating databases in DIM system are translated into schemas of the CDM.

The translated schemas are not integrated to conform a global schema. Furthermore, no schema integration is performed, and several translated schemas of the databases are available for the user.

DIM is based on the assumption that a common host language is available for each one of the integrated database systems. A command in the query language of DIM is translated into a program in the common host language with embedded data manipulation commands of the database systems. A role played by a common host language, is central in DIM to easily embed different database systems commands in one program in the common host language.

To construct a DIM system, the first step is to find the common host language among the participating database systems. The next step is to choose a CDM, and a database query language. The CDM should be rich enough to represent schemas of the databases for the schema translation, and the query language should be such that queries in the language can be translated into queries of the database systems.

DIM method was applied for integration of object-oriented and relational database systems. In this particular case of application of DIM, the ObjectStore C++ was discovered to be a suitable common host language. We chose relational modelling of data as the CDM, and SQL as the query language for the particular case. In addition to the simplicity of the relational model, it is more probable that the number of relational database systems participating in the integration be more than the number of object-oriented database systems in an organisation. To represent the translated schemas of the object-oriented database in the integration, virtual tables are suitable mechanism. By means of the virtual tables, the user views the object-oriented database as a set of relational tables.

In this thesis little mention was made concerning problems of global transaction management, and semantic heterogeneity of participating databases in DIM system, although highly important in HDBS operations.

Problem of transaction management for atomicity of the global transaction is thought to be solved by updating only one database for each update command. In the case of participation of two database systems such as object-oriented and relational database systems, the user is allowed to update virtual or real tables, not both kinds of the tables. Thus, the problem is given over to each transaction management system of individual databases. At this stage, we didn't consider the problem of consistent updates for replicated data in the databases, because the update problem by itself is very difficult, and it does not fit in the thesis.

Although the translated schemas are not integrated, semantics of data in the translated schemas are very important. The problem of semantic heterogeneity can be solved by the GDBA. In the case of object-oriented and relational databases, the GDBA defines the virtual tables and names the columns of those virtual tables by himself, or by cooperating with every local DBAs. The GDBA also establishes relationships between the two kinds of tables. Construction of the virtual tables information (VTI), that is determining which methods and access paths should be used to represent the real presentation of columns of virtual tables is also left to the GDBA.

The DIM system in the case of object-oriented and relational database, generates a program in regard to a query that is made against virtual and real tables. If the query merely includes virtual tables, the generated program will be a program that contains the access paths and methods to access the persistent objects of the object-oriented database.

The program can be used as a function in another program for accessing the persistent objects.

Apart from the long response time, because of the time for generating and compiling a program, a system based on DIM method is a practical system for integration of heterogeneous databases, if the common host language be available among the database systems. DIM system can be run on a powerful and fast computer system so that the response time be decreased.

Bibliography:

[1] Agrawal R., Gehani N. H., "ODE (Object Database and Environment): The Language and the Data Model," Gupta R., Horowitz E. (Ed.), Object-Oriented Databases with Applications to CASE, Networks, and VLSI CAD, Prentice Hall, New Jersey, 1991.

[2] Atzeni P., "Languages For Databases," Cioni G., Salwicki A. (Ed.), Advanced Programming Methodologies, London, Academic Press Limited, 1989.

[3] Beynon-Davies P., Relational Database Design, Blackwell Scientific Publications, Oxford, 1992.

[4] Blair G., et al., Object-Oriented Languages, Systems and Applications, PITMAN, Great Britain, 1991.

[5] Bright M. W. et al., "A Taxonomy and Current Issues in Multidatabase Systems, " IEEE Computer, March 1992, PP 50-60.

[6] Budd T., An Introduction to Object-oriented Programming, Addison-Wesley, Massachusetts, 1991.

[7] Cattel R. G. G., Object Data Management: Object-Oriented and Extended Relational Database Systems, Addison-Wesley, Massachusetts, 1991.

[8] Ceri S., Pelagatti G., Distributed Databases: Principles & Systems, McGRAW-HILL, Singapore, 1985.

[9] Date C. J., An Introduction To Database Systems, Fifth Ed., Vol. 1, Addison-Wesley, Massachusetts, 1990.

[10] Dawson J., " A Family of Models" , Byte Sept 1989, PP 277-286.

[11] Duchene H., et al., "VODAK Kernel Data Model," Dittrich K. R. (Ed.), Lecture Notes in Computer Sciences: Advances in Object-Oriented Database Systems: 2nd International Workshop on Object-Oriented Database Systems, Springer-Verlag, Berlin, 1988.

[12] Elmasri R., Navathe S. B., Fundamentals of Database Systems, Benjamin/ Cunnings, Redwood City, Calif., 1989.

[13] Gupta R. et al., "The Development of a Framework for VLSI CAD," Gupta R., Horowitz E. (Ed.), Object-Oriented Databases with Applications to CASE, Networks, and VLSI CAD, Prentice Hall, New Jersey, 1991.

[14] Harris C., Duhl J, "Object SQL," Gupta R., Horowitz E. (Ed.), Object-Oriented Databases with Applications to CASE, Networks, and VLSI CAD, Prentice Hall, New Jersey, 1991.

[15] Heiler S., Zdonik S., "Views, Data Abstraction, and Inheritance in the Fugue Data Model," , Dittrich K. R. (Ed.), Lecture Notes in Computer Science: Advances in Object-Oriented Database Systems, Springer-Verlag, New York, 1988.

[16] Horowitz E., Wan Q., "An Overview of Existing Object-Oriented Database Systems", Gupta R., Horowitz E. (Ed.), Object-Oriented Databases with Applications to CASE, Networks, and VLSI CAD, Prentice Hall, New Jersey, 1991.

[17] Hsiao D. K., "The Object-oriented Database Management: A Tutorial on its Fundamentals", Proceedings of the Second Far-East Workshop on Future Database Systems, April, 1992.

[18] Hughes J., Object-Oriented Databases, Prentice Hall, New York, 1991.

[19] Kent W., "Limitations of Record-Based information Models," ACM Trans. Database Syst., March 1979, Vol. 4, No. 1, PP 107-131.

[20] Khoshafian S., Abnous R., Object Orientation: Concepts, Language, Databases, User Interfaces, John Wiley, New York, 1990.

[21] Kim V., "On Unified Relational and Object-Oriented Database System, " Potter J., et al. (Ed.), Technology of Object-Oriented Languages and Systems, Proceedings of Sixth International Conference TOOLS: Sydney 1992, Prentice Hall, New York, 1991, PP 5-17.

[22] Korson T., McGregor J. D., "Understanding Object-Oriented: A Unifying Paradigm," Communications of ACM, Sept. 1990, Vol 33, No. 9, PP 41-60.

[23] Litwin W., et al., "Interoperability of Multiple Autonomous Databases," ACM Computing Surveys, Vol. 22, No. 3, Sept. 1990, PP 267-293.

[24] Litwin W., Abdellatif A., "Multidatabase Interoperability ," IEEE Computer, Dec. 1986, PP 10-18.

[25] Motro A., "Superviews: Virtual Integration of Multiple Databases ," IEEE Trans. Soft. Eng., Vol. 13, No. 7, July 1987, PP 785-798.

[26] Papazoglou M., Valder W., Relational Database Management: A Systems Programming Approach, Prentice Hall International, New York, 1989.

[27] Parsaye K. et al., Intelligent Databases: Object-Oriented, Deductive, Hypermedia Technologies, John Wiley, New York, 1989.

[28] Ram S., "Heterogeneous Distributed Database System," IEEE Computer, Dec. 1991, PP 7-10.

[29] Rambaugh J. et al., Object-Oriented Modelling and Design, Prentice Hall, London, 1991.

[30] Saltor F. et al., "Suitability of data models as a canonical models for federated databases," ACM Sigmod Record, Vol. 20, No. 4, Dec. 1991, 44-48.

[31] Sheth A. P., "Semantic Issues in Multidatabase Systems," ACM Sigmod Record, Vol. 20, No. 4, Dec. 1991, PP 5-9.

[32] Sheth A. P., Larson J. A., " Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases," ACM Computing Surveys, Vol. 22, No. 3, Sept. 1990, PP 183-236.

[33] Smith K. E., Zdonik S. B., "Intermedia: A Case Study of the Differences Between Relational and Object-Oriented Database Systems," Proc. ACM Conf. OOPSLA, Portland, OR., 1986, PP 1-16.

[34] Tari Z., "Interoperability between Database Models, " IFIP, DS-5 Semantics of Interoperable Database System, Lorne, Victoria, Australia, Nov., 1992, Vol. 1, PP 96-113.

[35] Thompson J. P., Data with Semantics, Data Models and Data Management, Van Nostrand Reinhold, New York, 1989.

[36] Ullman J. D., Principles of Database and Knowledge-Based Systems, Vol 1, Computer Science Press, 1988.

[37] Winblad A. L., et al., Object-Oriented Software, Addison-Wesley, Massachusetts, 1990.

Appendices

Appendix

A

Sample Pro*C++ Programs

The following two examples are based on the SELECT statements (Section 6.3) of the implementation chapter. The SELECT statements are exactly the same as those that were used in the implementation chapter. Also, the same schemas (Figure 16), and VTI (Table 4) are used. In the following of each query, a Pro*C++ program which is generated as the result of execution of the query, is given. In fact, the Pro*C++ program is intermediate result of the query execution. The program must be preprocessed, and tuned for desired output. In Pro*C++ programs, generated variables for relational, and object-oriented databases have the prefixes "AAA" , "OOO" , respectively.

Example (1) SELECT:

```
SELECT  PARTS.PART_NUMBER, COLOR, WEIGHT, PART_DESC
        FROM
        PARTS, RELPARTS_A
        WHERE
        PARTS.PART_NUMBER = RELPARTS_A.PART_NUMBER ;
```

A generated Pro*C++ as the result of execution of the SELECT statement is:

Generated Program	Comments
#include <iostream.h> #include <ostore/ostore.hh> #include <ostore/coll.hh> #include <string.h> #include <stdio.h> #include <ctype.h> #include "StrComp.h" #include "PARTS.hh"	*Declaration of Heder Files.*
os_Set<PARTS*> *PARTS_extent = 0 ;	*Using ObjectStore class Extension.*
EXEC SQL BEGIN DECLARE SECTION;	*Beginning of ORACLE host variables declaration.*
VARCHAR uid[20];	*User ID. for login to ORACLE RDBMS.*
VARCHAR pwd[20];	*User password.*
VARCHAR AAAPART_NUMBER[30];	*Host Variable.*
VARCHAR AAAPART_DESC[30];	*Host Variable.*
EXEC SQL END DECLARE SECTION;	*End of the variables declaration.*
EXEC SQL DECLARE ORACUR CURSOR FOR SELECT PART_NUMBER, PART_DESC FROM RELPARTS_A ;	*Cursor declaration for real tables used in the query.*
EXEC SQL INCLUDE sqlca.h ;	*SQL communication area needed for event, and exception handling.*
char* HostAttrs = ":AAAPART_NUMBER,:AAAPART_DESC" ;	*Host variables appear in the sequence which their associated columns used in the cursor declaration .*

main() {
os_Set<PARTS*> &selected_PARTS =
os_Set<PARTS*>::create(database::get_transient_database ());	

Generated Program	Comments

```
PARTS *OOOPARTS ;
database  *PARTSdb  =  database::open(argv[1]);
OS_BEGIN_TXN(tx1, 0, transaction::update)
PARTS_extent = (os_Set<PARTS*>*) (
  PARTSdb->find_root
    ("PARTS_extent_root")->get_value() );        instances
                                                 of the PARTS class were
                                                 saved with the root
                                                 "PARTS_extent_root"

selected_PARTS  =  PARTS_extent->query           Selection of all instances of
            ( "PARTS*","1",PARTSdb);             the PARTS class.

os_Cursor<PARTS*>
      PARTSObCur (selected_PARTS) ;              ObjectStore
                                                 cursor declaration for all
                                                 instances of the PARTS class .

int  TupleCount = 0;                             A counter for selected
                                                 tuples .

                                                 Header for desired output .
printf("\n    PART_NUMBER  COLOR  WEIGHT  PART_DESC\n") ;
printf("---------------------------------------------------------------------- \n");

strcpy((char *)uid.arr,"/");                     For automatic login to
uid.len = strlen((char *)uid.arr);               ORACLE RDBMS.

EXEC SQL WHENEVER SQLERROR STOP;
EXEC SQL OPEN ORACUR;

while(1)      {
 EXEC SQL FETCH ORACUR INTO
           :AAAPART_NUMBER,:AAAPART_DESC ;
```

Generated Program	Comments

```
if ( sqlca.sqlcode ) break;
AAAPART_DESC.arr[AAAPART_DESC.len] = '\0' ;        Puts null values at the
                                                   end of the string.
for(OOOPARTS = PARTSObCur.first();OOOPARTS;
      OOOPARTS = PARTSObCur.next()) {              Loop for ObjectStore
                                                   cursor.


    StrComp
      StrB((char *)OOOPARTS->GetPartNo());         Instantiating the class
    StrComp                                        "StrComp" for
      StrC((char *)AAAPART_NUMBER.arr);            Comparison of strings
                                                   (operator overloading).


  if( StrB == StrC ) {                             Condition for printing
    cout<<"\n" <<"     "<<                         the output.
          OOOPARTS->GetPartNo() <<" "<<
          OOOPARTS->color <<" "<<
          OOOPARTS->GetWeight() <<" "<<
          (char*)AAAPART_DESC.arr << "\n";
    TupleCount++;
   }
  } /* for OS Cursor */                            End of ObjectStore
                                                   cursor Loop.

  } /* for Relational Cursor */                    End of ORACLE
                                                   cursor Loop.

 cout << "\n "<< TupleCount << " Row(s) selected \n";
OS_END_TXN(tx1)
return(0);
 }
```

Example (1) SELECT, after preprocessing with the C preprocessor:

```
/* File name & Package Name */
struct sqlcxp
{
 unsigned short fillen;
        char  filnam[11];
};
static struct sqlcxp sqlfpn =
{
    10,
    "SAMPLE1.pc"
};


static unsigned long sqlctx = 0;


static struct sqlexd {
  unsigned long    sqlvsn;
  unsigned short   arrsiz;
  unsigned short   iters;
  unsigned short   offset;
  unsigned short   selerr;
  unsigned short   sqlety;
  unsigned short   unused;
         short  *cud;
  unsigned char   *sqlest;
         char   *stmt;
  unsigned char  **sqphsv;
  unsigned long   *sqphsl;
         short **sqpind;
  unsigned long   *sqparm;
  unsigned long  **sqparc;
  unsigned char   *sqhstv[2];
  unsigned long    sqhstl[2];
         short  *sqindv[2];
  unsigned long    sqharm[2];
  unsigned long   *sqharc[2];
} sqlstm = {4,2};
extern sqlcex(/*_ unsigned long *, struct sqlexd *, struct sqlcxp * _*/);
extern sqlcx2(/*_ unsigned long *, struct sqlexd *, struct sqlcxp * _*/);
extern sqlcte(/*_ unsigned long *, struct sqlexd *, struct sqlcxp * _*/);
extern sqlbuf(/*_ char * _*/);
extern sqlora(/*_ long *, void * _*/);

 static char *sq0001 =
"SELECT PART_NUMBER,PART_DESC FROM RELPARTS          ";
static int IAPSUCC = 0;
static int IAPFAIL = 1403;
static int IAPFTL  = 535;
```

```
extern    sqliem();
/* cud (compilation unit data) array */
static short sqlcud0[] =
{4,34,
2,0,1,53,9,69,0,0,0,1,0,
13,0,1,0,13,75,2,0,0,1,0,2,9,0,0,2,9,0,0,
};

#include <iostream.h>
#include <ostore/ostore.hh>
#include <ostore/coll.hh>
#include <string.h>
#include <stdio.h>
#include <ctype.h>
#include "StrComp.h"

#include  "PARTS.hh"

 os_Set<PARTS*> *PARTS_extent = 0 ;

/* SQL stmt #1
 EXEC SQL BEGIN DECLARE SECTION;
*/

 struct {
  unsigned short len;
  unsigned char arr[20];
  } uid;
/*
 VARCHAR uid[20];

*/
 struct {
  unsigned short len;
  unsigned char arr[20];
  } pwd;
/*
 VARCHAR pwd[20];

*/
 struct {
  unsigned short len;
  unsigned char arr[30];
  } AAAPART_NUMBER;
/*
 VARCHAR  AAAPART_NUMBER[30];

*/
 struct {
  unsigned short len;
  unsigned char arr[30];
  } AAAPART_DESC;
/*
 VARCHAR  AAAPART_DESC[30];
```

```
 EXEC SQL END DECLARE SECTION;
*/

/* SQL stmt #3
 EXEC SQL DECLARE ORACUR CURSOR FOR

 SELECT PART_NUMBER,PART_DESC

 FROM  RELPARTS  ;
*/

/* SQL stmt #4
 EXEC SQL INCLUDE sqlca.h;
*/
/*
 * $Header: sqlca.h \
 * 1041000.1.1120.1 93/04/16 15:30:12 agadre Generic<unix> \
 * $ sqlca.h
 */

/* Copyright (c) 1985,1986 by Oracle Corporation. */

/*
NAME
  SQLCA : SQL Communications Area.
FUNCTION
  Contains no code. Oracle fills in the SQLCA with status info
  during the execution of a SQL stmt.
NOTES
  If the symbol SQLCA_STORAGE_CLASS is defined, then the SQLCA
  will be defined to have this storage class. For example:

    #define SQLCA_STORAGE_CLASS extern

  will define the SQLCA as an extern.

  If the symbol SQLCA_INIT is defined, then the SQLCA will be
  statically initialized. Although this is not necessary in order
  to use the SQLCA, it is a good pgming practice not to have
  unitialized variables. However, some C compilers/OS's don't
  allow automatic variables to be init'd in this manner. Therefore,
  if you are INCLUDE'ing the SQLCA in a place where it would be
  an automatic AND your C compiler/OS doesn't allow this style
  of initialization, then SQLCA_INIT should be left undefined --
  all others can define SQLCA_INIT if they wish.

  If the symbol SQLCA_NONE is defined, then the SQLCA variable will
  not be defined at all.  The symbol SQLCA_NONE should not be defined
  in source modules that have embedded SQL.  However, source modules
  that have no embedded SQL, but need to manipulate a sqlca struct
  passed in as a parameter, can set the SQLCA_NONE symbol to avoid
  creation of an extraneous sqlca variable.
```

```
MODIFIED
   VGokhale  04/16/93 - Change longs to ints for DEC Alpha
   losborne  08/11/92 - No sqlca var if SQLCA_NONE macro set
  Clare     12/06/84 - Ch SQLCA to not be an extern.
  Clare     10/21/85 - Add initialization.
  Bradbury  01/05/86 - Only initialize when SQLCA_INIT set
  Clare     06/12/86 - Add SQLCA_STORAGE_CLASS option.
*/

#ifndef SQLCA
#define SQLCA 1

struct  sqlca
        {
        /* ub1 */ char    sqlcaid[8];
#if defined(__osf__) && defined(__alpha)
        /* b4  */ int     sqlabc;
        /* b4  */ int     sqlcode;
#else
        /* b4  */ long    sqlabc;
        /* b4  */ long    sqlcode;
#endif
        struct
          {
          /* ub2 */ unsigned short sqlerrml;
          /* ub1 */ char           sqlerrmc[70];
          } sqlerrm;
        /* ub1 */ char    sqlerrp[8];
#if defined(__osf__) && defined(__alpha)
        /* b4  */ int     sqlerrd[6];
#else
        /* b4  */ long    sqlerrd[6];
#endif
        /* ub1 */ char    sqlwarn[8];
        /* ub1 */ char    sqlext[8];
        };

#ifndef SQLCA_NONE
#ifdef  SQLCA_STORAGE_CLASS
SQLCA_STORAGE_CLASS struct sqlca sqlca
#else
        struct sqlca sqlca
#endif

#ifdef  SQLCA_INIT
        = {
            {'S', 'Q', 'L', 'C', 'A', ' ', ' ', ' '},
          sizeof(struct sqlca),
          0,
          { 0, {0}},
            {'N', 'O', 'T', ' ', 'S', 'E', 'T', ' '},
          {0, 0, 0, 0, 0, 0},
          {0, 0, 0, 0, 0, 0, 0, 0},
          {0, 0, 0, 0, 0, 0, 0, 0}
```

```
      }
#endif
      ;
#endif

#endif

/* end SQLCA */

 char*  HostAttrs = ":AAAPART_NUMBER,:AAAPART_DESC";

 char*  OtherHostAttrs = "";

 main()

 {

os_Set<PARTS*> &selected_PARTS =
os_Set<PARTS*>::create(database::get_transient_database ());

PARTS *OOOPARTS ;

 database  *PARTSdb = database::open(argv[1]);

 OS_BEGIN_TXN(tx1, 0, transaction::update)

 PARTS_extent = (os_Set<PARTS*>*) (
 PARTSdb->find_root("PARTS_extent_root")->get_value() );

 selected_PARTS = PARTS_extent->query(
  "PARTS*","1",PARTSdb);

 os_Cursor<PARTS*> PARTSObCur(selected_PARTS);

int   TupleCount = 0;

 printf("\n      PART_NUMBER COLOR WEIGHT PART_DESC\n");

 printf("-------------------------------------------------------------------------- \n");

 strcpy((char *)uid.arr,"/");

 uid.len = strlen((char *)uid.arr);

/* SQL stmt #5
 EXEC SQL WHENEVER SQLERROR STOP;
*/

/* SQL stmt #6
 EXEC SQL OPEN ORACUR;
*/
{
 sqlstm.stmt = sq0001;
 sqlstm.iters = (unsigned short)1;
```

```
sqlstm.offset = (unsigned short)2;
sqlstm.cud = sqlcud0;
sqlstm.sqlest = (unsigned char *)&sqlca;
sqlstm.sqlety = (unsigned short)0;
sqlcex(&sqlctx, &sqlstm, &sqlfpn);
if (sqlca.sqlcode < 0) exit(1);
}

while(1)

{

/* SQL stmt #7
 EXEC SQL FETCH ORACUR INTO
 :AAAPART_NUMBER,:AAAPART_DESC ;
*/
{
 sqlstm.iters = (unsigned short)1;
 sqlstm.offset = (unsigned short)13;
 sqlstm.cud = sqlcud0;
 sqlstm.sqlest = (unsigned char *)&sqlca;
 sqlstm.sqlety = (unsigned short)0;
 sqlstm.sqhstv[0] = (unsigned char *)&AAAPART_NUMBER;
 sqlstm.sqhstl[0] = (unsigned long)32;
 sqlstm.sqindv[0] = (short *)0;
 sqlstm.sqharm[0] = (unsigned long)0;
 sqlstm.sqhstv[1] = (unsigned char *)&AAAPART_DESC;
 sqlstm.sqhstl[1] = (unsigned long)32;
 sqlstm.sqindv[1] = (short *)0;
 sqlstm.sqharm[1] = (unsigned long)0;
 sqlstm.sqphsv = sqlstm.sqhstv;
 sqlstm.sqphsl = sqlstm.sqhstl;
 sqlstm.sqpind = sqlstm.sqindv;
 sqlstm.sqparm = sqlstm.sqharm;
 sqlstm.sqparc = sqlstm.sqharc;
 sqlcex(&sqlctx, &sqlstm, &sqlfpn);
 if (sqlca.sqlcode < 0) exit(1);
 }

 if ( sqlca.sqlcode ) break;

 AAAPART_DESC.arr[AAAPART_DESC.len] = '\0' ;
 for(OOOPARTS = PARTSObCur.first();OOOPARTS;OOOPARTS =
PARTSObCur.next()) {

 StrComp  StrB((char *)OOOPARTS->GetPartNo());

 StrComp  StrC((char *)AAAPART_NUMBER.arr);

 if(
 (   StrB  ==       StrC )
 )
 {
```

```
cout<<"\n" <<"      "<<OOOPARTS->GetPartNo() <<" "<<OOOPARTS->color <<"
"<<OOOPARTS->GetWeight() <<" "<<(char*)AAAPART_DESC.arr << "\n";

TupleCount++;
}
      } /* for OS Cursor */

} /* for Relational Cursor */

cout << "\n "<< TupleCount << " Row(s) selected \n";

OS_END_TXN(tx1)

return(0);

}
```

Example (2) SELECT:

```
SELECT  *
        FROM  PARTS
        WHERE  PART_NUMBER  =  'p5'  ;
```

In the SQL example, only the virtual table PARTS is queried. The resulted Pro*C++ is a pure ObjectStore C++ code.

A generated Pro*C++ program as the result of execution of the query is:

```
os_Set<PARTS*> *PARTS_extent = 0 ;
main()  {
   os_Set<PARTS*> &selected_PARTS =
       os_Set<PARTS*>::create(database::get_transient_database ());
   PARTS *OOOPARTS ;
   database  *PARTSdb = database::open(argv[1]);
   OS_BEGIN_TXN(tx1, 0, transaction::update)
   PARTS_extent = (os_Set<PARTS*>*) (
```

```
        PARTSdb->find_root("PARTS_extent_root")->get_value() );

    selected_PARTS = PARTS_extent->query( "PARTS*","1",PARTSdb);
    os_Cursor<PARTS*> PARTSObCur(selected_PARTS);
    int  TupleCount = 0;
    printf("\n   PART_NUMBER   PART_NAME    COLOR  WEIGHT  CITY
                                                          \n");
    printf("-------------------------------------------------------------------------- \n");
    for(OOOPARTS = PARTSObCur.first();OOOPARTS; OOOPARTS =
                PARTSObCur.next()) {
       StrComp        StrA((char *)"p5");
       StrComp        StrC((char *)OOOPARTS->GetPartNo());
       if( StrC  ==  StrA )
         {

                cout<<"\n" << OOOPARTS->GetPartNo() <<"  "<< OOOPARTS->
                part_name  <<"  "<<OOOPARTS->color <<"  "<<OOOPARTS->
                GetWeight() <<"  "<<OOOPARTS->city <<"  "<< "\n";

                TupleCount++;
                }
     } /* for OS Cursor */
cout << "\n "<< TupleCount << " Row(s) selected \n";
OS_END_TXN(tx1)

return(0);

}
```

Appendix

B

Acronyms

The following is the list of acronyms introduced, or mentioned in the text.

ANSI	american national standards institute
CDM	Common Data Model
DB	database
DBA	database administrator
DBI	database integrator
DBMS	database management system
DCL	data control language
DDB	distributed database
DDBMS	distributed database management system
DDBS	distributed database system
DDL	data definition language
DIM	Database Integration Methodology
DML	data manipulation language
GDBA	global database administrator
HDB	heterogeneous database
HDBMS	heterogeneous database management system
HDBS	heterogeneous database system
OODBMS	object-oriented database management system
OODB	object-oriented database

OODBS	object-oriented database system
RDBMS	relational database management system
SQL	structured query language
VTI	Virtual Tables Information
YACC	yet another compiler-compiler